Application Screening 1.0 Component Specification

1.Design

The TopCoder Software application methodology requires Project Managers to follow a strict guideline for creating requirement specifications. This component aids Project Managers in creating their UML models by creating warnings when the proper format is not followed. The component can be used to introspect the XMI submitted by the Project Manager and validate numerous rules. It also provides generation of reports from the introspected data.

The Application Screening component provides following features:

- Introspecting XMI data. The XMI Parser component is used to read and parse the XMI submission. In the current release only use case and activity diagram information is retrieved. Note that the XMI Parser component was extended to implement parsing of the activity diagrams and the activity graph information. Details of this extension and required modifications can be found in the altered documentation for the XMI Parser included with this submission.

- Pluggable validation. The component supports pluggable validation. The default set of validators implementing validation logic documented in the RS is supported. Custom validators can be easily added either directly or using a Factory interface which can be combined with static configuration based on the Configuration Manager component.

- Validation as a way of report generation. Not all validators are used to find errors in the application design, some of them are intended only to generate reports based on the submission data. For example, the *ActivityDiagramPathReportGenerator* class generates a report of all unique execution paths through an activity diagram. Note that this class can also be considered as a validator which always returns true. Some validators actually perform both functions simultaneously (validation and report generation); they can return heterogeneous data containing both error information and additional reports.

- Pluggable output formatting. Validation output (both information on errors and additional reports) can be formatted using pluggable formatters. Formatters can also be created using a Factory or from static configuration data. The component provides two default formatters: for text and XML representation.

1.1Design Patterns

- Validators and formatters are implemented as pluggable **Strategies**.
- **Abstract Factory** design pattern is used to instantiate formatters and validators. The concrete factory loading information from static component configuration is provided.
- *UseCaseDiagramValidator* and *ActivityDiagramValidator* abstract classes provide simple **Templates** for validation of use case and activity diagrams (refer to class docs for more details).
- *ValidationManager* provides a simple **Facade** to the whole component.

1.2Industry Standards

UML and **XMI** are active standards managed by OMG. Current specifications can be downloaded from www.uml.org. Note that the Application Screening doesn't access the XMI file directly but only through the XMI Parser component. All modifications to the current XMI standard should be reflected in the XMI Parser.

1.3Required Algorithms

Essentially no complex algorithms are needed to develop this component. For convenience of the component developer this chapter provides basic description of each validation algorithm defined in the component. Note that the code given is a pseudo-code not the actual Java code; for simplicity of explanation it may contain some constructs (like foreach or global variables) not existing in the Java language. It also omits all conversions between collections (like Lists) and static array types; the developer is required to provide such conversion whenever it will be needed.

UseCaseDiagramValidator

This class defines a basic template for validation of all use case diagrams in the submission.

```
ValidationOutput[] validateSubmission(Submission s)
{
    List retList = new ArrayList();

    if (s.getUseCaseDiagrams().isEmpty())
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR,
                "at least one use case diagram should be defined"
            )
        );
    }

    foreach (UMLUseCaseDiagram d in s.getUseCaseDiagrams())
    {
        retList.add(validateUseCaseDiagram(d, s));
    }

    return retList;
}
```

ActivityDiagramValidator

This class defines a basic template for validation of all activity diagrams in the submission.

```
ValidationOutput[] validateSubmission(Submission s)
{
    List retList = new ArrayList();

    if (s.getUseCaseDiagrams().isEmpty())
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR,
                "at least one activity diagram should be defined"
            )
        );
    }
```

```
    foreach (UMLActivityDiagram d in s.getActivityDiagrams())
    {
        retList.add(validateActivityDiagram(d, s));
    }

    return retList;
}
```

DefaultUseCaseDiagramValidator

This class should validate that each use case diagram has at least one use case and at least one actor. All use cases and actors should have non-empty names.

```
ValidationOutput[] validateUseCaseDiagram(
    UMLUseCaseDiagram d, Submission s
)
{
    List retList = new ArrayList();

    boolean hasConcreteUseCase = false;

    foreach (UMLUseCase uc in d.getUseCases())
    {
        if (!uc.isAbstract())
        {
            hasConcreteUseCase = true;
        }
        else
        {
            retList.add(new ValidationOutput(
                        ValidationOutputType.ERROR, d, uc,
                        "the use case diagram has an abstract use case"
                )
            );
        }

        if (uc.getName().trim().equals("")) {
            retList.add(new ValidationOutput(
                        ValidationOutputType.ERROR, d, uc,
                        "the use case has empty name"
                )
            );
        }
    }

    if (!hasConcreteUseCase)
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d,
                "the use case diagram is missing a concrete use case"
            )
        );
    }

    if (d.getActors().isEmpty())
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d,
                "the use case diagram is missing an actor"
            )
        );
    }

    foreach (UMLActor a in d.getActors())
    {
        if (a.getName().trim().equals("")) {
            retList.add(new ValidationOutput(
                        ValidationOutputType.ERROR, d, a,
```

```
                        "the actor has empty name"
                    )
                );
            }
        }

        return retList;
}
```

DefaultActivityDiagramValidator

This class should validate that each activity diagram has at least one initial state and at least one final state. All final and initial states should have non-empty names.

```
ValidationOutput[] validateActivityDiagram(
    UMLActivityDiagram d, Submission s
)
{
    List retList = new ArrayList();

    if (d.getInitialStates().isEmpty())
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d,
                "the activity diagram is missing an initial state"
            )
        );
    }

    if (d.getFinalStates().isEmpty())
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d,
                "the activity diagram is missing a final state"
            )
        );
    }

    foreach (UMLState st in d.getInitialStates())
    {
        if (st.getName().trim().equals("")) {
            retList.add(new ValidationOutput(
                        ValidationOutputType.ERROR, d, st,
                        "the initial state has empty name"
                    )
                );
            }
    }

    foreach (UMLState st in d.getFinalStates())
    {
        if (st.getName().trim().equals("")) {
            retList.add(new ValidationOutput(
                        ValidationOutputType.ERROR, d, st,
                        "the final state has empty name"
                    )
                );
            }
    }

    return retList;
}
```

DefaultUseCaseDiagramNamingValidator

This class should validate that each use case has at least one corresponding activity diagram. The diagram should be named as "<use case name> Activity - <optional part>" or "<use case name> Activity Diagram - <optional part>" . The optional part will be

present only if there are at least two activity diagrams for this use case. The naming comparison is case insensitive.

```
ValidationOutput[] validateUseCaseDiagram(
    UMLUseCaseDiagram d, Submission s
)
{
    List retList = new ArrayList();

    foreach (UMLUseCase uc in d.getUseCases())
    {
        String useCaseName = uc.getName().trim().toLowerCase();

        int cntDiagrams = 0;
        boolean hasExactMatch = false;

        foreach (UMLActivityDiagram ad in s.getActivityDiagrams())
        {
            String adName = ad.getName().trim().toLowerCase();

            if (!adName.startsWith(useCaseName)) continue;

            cntDiagrams++;

            String rem = adName.substring(useCaseName.length()).trim();

            if (rem.startsWith("activity diagram"))
                rem = rem.substring("activity diagram".length());
            else if (rem.startsWith("activity"))
                rem = rem.substring("activity".length());
            else
                continue;

            rem = rem.trim();
            if (rem.length() != 0 && !rem.startsWith("-"))
                continue;

            cntDiagrams++;
            if (rem.length() == 0)
                hasExactMatch = true;
        }

        if (cntDiagrams == 0)
        {
            retList.add(new ValidationOutput(
                    ValidationOutputType.ERROR, d, uc,
                    "the use case doesn't have any "+
                    "corresponding activity diagrams"
                )
            );
        }

        if (!hasExactMatch && cntDiagrams == 1)
        {
             retList.add(new ValidationOutput(
                    ValidationOutputType.ERROR, d, uc,
                    "the use case with a single "+
                    "corresponding activity diagram should not have "+
                    "the optional part in the activity diagram name"
                )
            );
        }
    }

    return retList;
}
```

DefaultActivityDiagramNamingValidator

This class should validate that each activity diagram has at least one corresponding use

case. The activity diagram name should start from a use case name following the "Activity" keyword and possibly an optional part. The comparison is case insensitive.

```
ValidationOutput[] validateActivityDiagram(
    UMLActivityDiagram d, Submission s
)
{
    List retList = new ArrayList();

    String adName = d.getName().trim().toLowerCase();
    int idx = adName.indexOf("activity");

    if (idx == -1)
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d,
                "the activity diagram name is missing " +
                "the Activity keyword"
            )
        );

        return retList;
    }

    String matchWith = adName.substring(0, idx).trim();

    boolean matchFound = false;
    foreach (UMLUseCaseDiagram d in s.getUseCaseDiagrams())
    foreach (UMLUseCase uc in d.getUseCases())
    {
        if (uc.getName().trim().toLowerCase().equals(matchWith))
        {
            matchFound = true;
        }
    }

    if (!matchFound)
    {
        retList.add(new ValidationOutput(
                ValidationOutputType.ERROR, d, uc,
                "the activity diagram doesn't have any "+
                "corresponding use cases"
            )
        );
    }

    return retList;
}
```

ActivityDiagramPathReportGenerator

This class doesn't perform any validation of the activity diagram; instead it generates a report of all unique paths throughout the diagram. Each path will be assigned a unique path number; the path will be represented by a sequence of activity diagram nodes and transitions. Guard names for transitions will be used to identify edges; if some transition doesn't have the guard name, the state name only will be used to identify the transition.

NOTE: The activity graph for an activity diagram may contain loops. To resolve such situation all unique paths will require that no activity graph state will be traversed more than twice (two visits of the same state i.e. a single loop traversal is allowed).

The algorithm described below uses depth-first search to traverse the activity graph. The current path is stored in a *LinkedList* structure which provides efficient implementation for addLast and removeLast operations.

```
// "global" path count variable
int pathCount;
```

```
ValidationOutput[] validateActivityDiagram(
    UMLActivityDiagram d, Submission s
)
{
    List retList = new ArrayList();

    pathCount = 0;

    // we run DFS for all transitions from the initial state
    foreach(UMLState st in d.getInitialStates())
    {
        foreach(UMLTransition t in d.getTransitionsFrom(st.getId()))
        {
            LinkedList transitions = new LinkedList();

            transitions.add(t);

            dfs(d, t.getToId(), transitions, retList);
        }
    }

    return retList;
}

// this "worker" method performs DFS
void dfs(UMLActivityDiagram d, String currentNodeId,
         LinkedList transitions, List retList)
{
    // check if we visited the final state
    // the developer is encouraged to optimize
    foreach(UMLState st in d.getFinalStates())
    {
        if (st.getId().equals(currentNodeId))
        {
            // the current state is final
            // add one more path record
            pathCount++;

            retList.add(new ValidationOutput(
                    ValidationOutputType.REPORT, d,
                    buildPathReport(d, pathCount, transitions)
                )
            );

            return;
        }
    }

    // not the final state
    // we should try all transitions
    // don't visit the same state third time
    foreach(UMLTransition t in d.getTransitionsFrom(currentNodeId))
    {
        int cntVisited = 0;

        foreach(UMLTransition prev in transitions)
        {
            if (prev.getFromId().equals(t.getToId()))
                cntVisited++;

            if (prev.getToId().equals(t.getToId()))
                cntVisited++;
        }

        if (cntVisited <= 1)
        {
            // the transition is new, try it
            transitions.addLast(t);

            dfs(d, t.getToId(), transitions, retList);
```

```
            transitions.removeLast();
        }
    }

    return;
}

// this methods builds the details string (XML) for the given
// list of transitions
String buildPathReport(UMLActivityDiagram d, int pathNumber,
                       List transitions)
{
    StringBuffer buf = new StringBuffer();

    buf.append("<Path number='" + String.valueOf(pathNumber) + "'>");
    foreach(UMLTransition t in transitions)
    {
        buf.append("<Node>");
        foreach(UMLState st in d.getAllStates())
        {
            if (st.getId().equals(t.getFromId()))
            {
                buf.append(st.getName());
                break;
            }
        }
        if (!t.getGuardConditionName().equals(""))
        {
            buf.append(" - ");
            buf.append(t.getGuardConditionName());
        }
        buf.append("</Node>");
    }
    buf.append("</Path>");

    return buf.toString();
}
```

XML Schema for XML formatter output

Output of the *XMLValidationOutputFormatter* should conform to the following XML schema.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:attribute name="id" type="xs:string" use="required"/>
<xs:attribute name="type" type="xs:string" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:complexType name="error" mixed="true">
  <xs:any minOccurs="0" maxOccurs="unbounded"/>
</xs:complexType>
<xs:complexType name="report" mixed="true">
  <xs:any minOccurs="0" maxOccurs="unbounded"/>
</xs:complexType>
<xs:complexType name="element with errors">
  <xs:sequence>
    <xs:element name="error" type="error" minOccurs="1"
                maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
  <xs:attribute ref="type"/>
  <xs:attribute ref="name"/>
</xs:complexType>
<xs:complexType name="element with reports">
  <xs:sequence>
    <xs:element name="report" type="report" minOccurs="1"
                maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
  <xs:attribute ref="type"/>
```

```
  <xs:attribute ref="name"/>
</xs:complexType>
<xs:complexType name="diagram with errors">
  <xs:sequence>
    <xs:element name="error" type="error" minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="element" type="element with errors" minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
  <xs:attribute ref="type"/>
  <xs:attribute ref="name"/>
</xs:complexType>
<xs:complexType name="diagram with reports">
  <xs:sequence>
    <xs:element name="report" type="report" minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="element" type="element with reports" minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
  <xs:attribute ref="type"/>
  <xs:attribute ref="name"/>
</xs:complexType>
<xs:element name="validation">
  <xs:complexType name="errors">
    <xs:sequence>
      <xs:element name="error" type="error" minOccurs="0"
                  maxOccurs="unbounded"/>
      <xs:element name="diagram" type="diagram with errors" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="reports">
    <xs:sequence>
      <xs:element name="report" type="report" minOccurs="0"
                  maxOccurs="unbounded"/>
      <xs:element name="diagram" type="diagram with reports"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

1.4Component Class Overview

All public classes and interfaces are listed in this section.

ValidationManager:

This class provides a simple facade to the whole component. It manages a set of validators for the XMI data and a formatter to format the validation output. It also provides several overloaded API methods to perform the actual validation and return the formatted output.

The XMI Parser component is used to perform parsing of the XMI data.

The class can be configured directly via the constructor parameters or using a Configuration Manager. If static configuration is used, the following properties will be read:

1. "factory_class" to load the class of a *SubmissionValidatorFactory* that will be used to create the validators and the formatter. This property is mandatory.
2. "factory_namespace" to load the namespace for the factory. This property is optional. If the property is not specified the factory will be created using the default constructor, otherwise using the constructor with a namespace (String) parameter.

Thread-Safety: this class is immutable and therefore thread-safe. Note that this class doesn't keep any private references to an *XMIParser* or a *DefaultXMIHandler* instance because these classes are not thread-safe. Instead a new parser instance will be created each time the parse() method is called

Submission:

This class encapsulates parsed submission data needed for the validation and defines no additional logic. In the current design, validators need only information on use case and activity diagrams, therefore this class keeps collections of *UMLActivityDiagrams* and *UMLUseCaseDiagrams* (both classes are defined in the XMI Parser component).

Thread-Safety: this class is immutable and therefore thread-safe.

(abstract) SubmissionValidator:

This class defines an abstract submission validator. It declares an abstract *validateSubmission*() method which returns a static array of *ValidationOutput* items. Each item in the returned array is either an error message or a report related to the validated XMI. Implementation of this method must be provided by all concrete validators.

The submission is considered valid if no ERROR items were generated by the *validateSubmission*() method.

Thread-Safety: this class is stateless and therefore thread-safe. The component assumes that all concrete implementations of the *validateSubmission* method will be thread-safe.

ValidationOutputType:

This class is a type safe enumeration of possible types of the validation output. Two types are currently defined: ERROR - for error messages related to problems with the submission and REPORT - for reports generated based on the introspected XMI data. Additional types can be defined via the inheritance.

Thread-Safety: This class is immutable and therefore thread-safe.

ValidationOutput:

This class represents a single item of the validator output. Note that a single validator can generate many different *ValidationOutput* items for a single submission. Each validation output has a type (ERROR or REPORT), a related UML diagram, a related UML element and a string of details. Note that the UML element and even the UML diagram may be unspecified (for example, an error is related to the whole submission but not to a particular diagram/element); if so null values will be returned by appropriate class getters.

A string of details may be a simple text string or it may have complex structure (as for the activity diagram paths report). The details string and the validation output should be passed to some formatter class to generate a comfortable human readable representation.

Thread-Safety: This class is immutable and therefore thread-safe.

(interface) ValidationOutputFormatter:

This interface declares a formatting entity. Formatters should be able to accept array of validation output containing both error information and reports and convert this data to a human readable representation. Each concrete formatter can define its own output

format. There are two default formatters provided by this component - for text and XML output.
Thread-Safety: all implementations of this interface are expected to be thread-safe.

(interface) SubmissionValidatorFactory:

This interface defines an abstract factory for submission validators and output formatters. The factory defines two methods: a method to create a set of validators to use and a method to create a formatter to use. The component provides the default factory implementation (*DefaultSubmissionValidatorFactory*) which instantiates validators and formatters based on the information loaded from the static component configuration.
Thread-Safety: all implementations of this interface are expected to be thread-safe.

DefaultSubmissionValidatorFactory implements SubmissionValidatorFactory:

This class provides the default factory implementation. This implementation loads static component configuration using the Configuration Manager and instantiates validators and formatters from this configuration via reflection API.
The following configuration properties are used by the factory:

1. "validators" - a multi-value property containing String names of validators to create. This property is mandatory (at least one validator should be configured).
2. <validator_name>+"_class" - a String property containing a class name for the validator <validator_name>. This property is mandatory (each <validator_name> mentioned in the "validators" should have this property).
3. <validator_name>+"_namespace" - a String property containing a ConfigManager namespace for the validator <validator_name>. This property is optional. If absent, then the validator will be created using the default constructor. If present, the validator will be created using the constructor with a namespace:String parameter.
4. "formatter_class" - a String property containing a class name for the formatter. This property is mandatory.
5. "formatter_namespace" - a String property containing a namespace for the formatter. This property is optional. If absent, the formatter will be created via the default constructor. If present, the formatter will be created via the constructor with a namespace:String parameter.

Thread-Safety: This class is immutable and therefore thread-safe.

(abstract) UseCaseDiagramValidator extends SubmissionValidator:

This abstract class defines validation template for use case diagrams. Validation of submission is implemented as validation of each use case diagram and aggregation of the validation output. All extensions of this class should only provide a primitive for validation of a single use case diagram, this primitive will be called for each use case diagram found in the submission and the resulting data will be aggregated automatically.
Additionally, this validator checks a mandatory condition that the submission has at least one use case diagram (if not, an ERROR is reported).
You may refer to the validation pseudo-code in the CS for more details.
Thread-Safety: This class is stateless and therefore thread-safe. All concrete implementations of this class should provide thread-safe implementation of the *validateUseCaseDiagram* method.

(abstract) ActivityDiagramValidator extends SubmissionValidator:

This abstract class defines validation template for activity diagrams. Validation of submission is implemented as validation of each activity diagram and aggregation of the validation output. All extensions of this class should only provide a primitive for validation of a single activity diagram, this primitive will be called for each activity diagram found in the submission and the resulting data will be aggregated automatically.
Additionally, this validator checks a mandatory condition that the submission has at least one activity diagram (if not, an ERROR is reported).

You may refer to the validation pseudo-code in the CS for more details.

Thread-Safety: This class is stateless and therefore thread-safe. All concrete implementations of this class should provide thread-safe implementation of the *validateActivityDiagram* method.

DefaultUseCaseDiagramValidator extends UseCaseDiagramValidator:

This class provides default validation of a use case diagram. The following conditions will be checked:

1. the diagram should have at least one use case;
2. the diagram should have at least one actor;
3. all use cases should be concrete (no abstract use cases are allowed);
4. all use cases should have non-empty names;
5. all actors should have non-empty names.

You may refer to the validation pseudo-code in the CS for more details.

Thread-Safety: This class is stateless and therefore thread-safe.

DefaultActivityDiagramValidator extends ActivityDiagramValidator:

This class provides default validation of an activity diagram. The following conditions will be checked:

1. the diagram should have at least one initial state;
2. the diagram should have at least one final state;
3. all initial states should have non-empty names;
4. all final states should have non-empty names.

You may refer to the validation pseudo-code in the CS for more details.

Thread-Safety: This class is stateless and therefore thread-safe.

DefaultUseCaseDiagramNamingValidator extends UseCaseDiagramValidator:

This class provides default naming validation of a use case diagram. The following conditions will be checked:

1. for each use case there should be at least one correspondent activity diagram. If the use case name is X then the activity diagram name can either be X Activity (- Y) or X Activity Diagram (- Y);
2. the optional part (- Y) can only be present if there are at least two activity diagrams corresponding to the use case.

The comparison is case insensitive. You may refer to the validation pseudo-code in the CS for more details.

Thread-Safety: This class is stateless and therefore thread-safe.

DefaultActivityDiagramNamingValidator extends ActivityDiagramValidator:

This class provides default naming validation of an activity diagram. It should check that the activity diagram has at least one corresponding use case. The activity diagram name should start from a use case name following the “Activity” keyword and possibly an optional part. The comparison is case insensitive.

You may refer to the validation pseudo-code in the CS for more details.

Thread-Safety: This class is stateless and therefore thread-safe.

ActivityDiagramPathReportGenerator extends ActivityDiagramValidator:

This class doesn’t perform any validation of the activity diagram; instead it generates a report of all unique paths throughout the diagram. Each path will be assigned a unique path number; the path will be represented by a sequence of activity diagram nodes and transitions. Guard names for transitions will be used to identify edges; if some transition doesn’t have the guard name, the state name only will be used to identify the transition.

The activity graph for an activity diagram may contain loops. To resolve such situation all unique paths will ensure that no activity graph state will be traversed more than twice (two visits i.e. a single loop traversal is allowed).

Each ValidationOutput item generated by this class will have the REPORT type. Each such item will correspond to a single unique path from some initial to some final state. The details string will contain an XML representation of the path in the following format:
<path number='${pathNumber}'>
<node>${initial node name} - ${guard transition name}</node>
<node>${next node name} - ${guard transition name}</node>
...
<path>
Note that the guard transition name is optional (it will be absent if the guard is empty); the final node is not included to the path. Number of returned ValidationOutput items will be equal to the number of unique paths in the diagram (discarding all paths containing any state more than twice).
Thread-Safety: this class is stateless and therefore thread-safe.

TextValidationOutputFormatter implements ValidationOutputFormatter:

This class is a concrete formatter for the validation output. It converts the data to the text format illustrated below (comments are given for informational purpose only).
Thread-Safety: This class is stateless and therefore thread-safe.

```
Format of the output data:
OutputType: [report|error]
// only if diagram reference is not empty
// diagramType is diagram.getDiagramType()
// diagramName is diagram.getDiagramName()
// diagramId is diagram.getDiagramId()
Diagram: diagramType diagramName(diagramId)
// only if element reference is not empty
// elementType is element.getElementType()
// elementName is element.getElementName()
// elementId is element.getElementId()
Element: elementType elementName(elementId)
Details: details string
```

XMLValidationOutputFormatter implements ValidationOutputFormatter:

This class is a concrete formatter for the validation output. It converts the data to the XML format illustrated below (comments are given for informational purpose only).
Thread-Safety: This class is stateless and therefore thread-safe.
Format of the output data:

```
<validation>
<errors>
<diagram id="$diagramId" type="$diagramType" name="$diagramName">
<element id="$elementId" type="$elementType" name="$elementName">
<error>details string</error>
<error>... more errors for this diagram/element</error>
</element>
... // more elements and diagram level errors
</diagram>
... // more diagrams and top level errors
</errors>
<reports>
<diagram id="$diagramId" type="$diagramType" name="$diagramName">
<element id="$elementId" type="$elementType" name="$elementName">
<report>details string</report>
<report>... more reports for this diagram/element</report>
</element>
... // more elements and diagram level reports
</diagram>
... // more diagrams and top level reports
</reports>
</validation>
```

Note that formatting of the data requires some aggregation:

1. all report information should be placed under <reports> node and all error information should be placed under <errors> node;
2. all errors/reports for a particular diagram should be grouped within the XML node for

this diagram;

3. all errors/reports for a particular element should be grouped within the XML node for this element (which is placed within the XML node for the corresponding diagram).

1.5Component Exception Definitions

All exceptions thrown by the component are listed below. For the details on exceptions thrown by a particular method refer to the method documentation.

java.lang.IllegalArgumentException:

This exception will be thrown to indicate that a method has been passed an illegal or inappropriate argument. Note that according to the new guidelines this exception will also be thrown whenever a null is passed to a method which can't handle the null parameter.

com.topcoder.util.xmi.parser.XMIParserException:

This exception will be propagated from the XMI Parser component if any problem happens while parsing the validated XMI input (IO problem, incorrect input format etc).

com.topcoder.apps.screening.application.specification.ConfigurationException:

This exception will be thrown to indicate any problems in the component configuration, such as missing a mandatory configuration property or failure to create a validator/formatter entity via reflection,
The exception will wrap all exceptions thrown by ConfigManager or reflection API.

com.topcoder.apps.screening.application.specification.FormatterException:

This exception will be thrown by *ValidationOutputFormatter* instances to indicate any problems in the format() method which prevent correct formatting of the given validation output. This exception is currently not thrown by the default formatters; it is reserved for the future use.

1.6Thread Safety

The component is thread-safe though this was not an original component requirement. The following measures were taken to ensure thread-safety:

- All abstract validator classes and concrete validators provided by the component are stateless and therefore thread-safe. If a custom validator is provided by the application developer, the component will assume that the custom validator validateSubmission method is thread-safe. No additional external locking will be used by the *ValidationManager* during the validate() call. In other words, using a non thread-safe validator with the component is possible, however it will affect the *ValidationManager* thread-safety.
- The previous comment can also be applied to the formatters. All concrete formatters provided by the component are stateless and therefore thread-safe. If a custom formatter is provided by the application developer, the component will assume that the custom formatter format method is thread-safe. No additional external locking will be used by the *ValidationManager* during the validate() call. In other words, using a non thread-safe formatter with the component is possible, however it will affect the *ValidationManager* thread-safety.
- *Submission, ValidationOutput* and *ValidationOutputType* classes are immutable and therefore thread-safe.
- *DefaultSubmissionValidatorFactory* class is immutable and therefore thread-safe. It is assumed that all custom factories provided by the application developer will also be

immutable or at least thread-safe.

- *ValidationManager* class is immutable and therefore thread-safe. Note to the developer: the *ValidationManager* uses the *XMIParser* and *DefaultXMIHandler* classes for parsing of XMI data. These classes (provided by the XMI Parser component) are not thread-safe. Two approaches can be chosen here to ensure thread-safety: create a new parser and handler instance each time the *parse* method is called (currently documented approach) or keep a reference to a single parser instance and lock the parser each time parsing needs to be performed. The concrete approach to use is up to the component developer.
- The parsed XMI is accessed in the read-only mode. It should be possible to share a single XMI file between two different threads or even processes/applications though such access is unlikely to be needed.

2.Environment Requirements

2.1Environment

Development language and compile target: **Java 1.4**

2.2TopCoder Software Components

Java Configuration Manager 2.1.4 is used to load the static component configuration

Java XMI Parser 1.0 is used to parse the validated XMI file

Java Type-Safe Enum 1.0 is used to support type-safe enumerations

Java Base Exception 1.0 provides base class for all custom exceptions

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3Third Party Components

None.

3.Installation and Configuration

3.1Package Name

com.topcoder.apps.screening.application.specification – main component package, defines the generic framework for validation and report generation
com.topcoder.apps.screening.application.specification.impl – main package for implementation classes (factories, validators and formatters); contains the default factory which sets up the component based on the static configuration
com.topcoder.apps.screening.application.specification.impl.validators – sub-package for the default validation functionality
com.topcoder.apps.screening.application.specification.impl.formatters – sub-package for the default formatting functionality

3.2Configuration Parameters

com.topcoder.apps.screening.application.specification.ValidationManager:

1. factory_class – required parameter; defines the fully qualified class name of the *SubmissionValidatorFactory* to use;
2. factory_namespace – optional parameter; defines the ConfigManager namespace for the factory. If absent, the default constructor will be used to create the factory.

com.topcoder.apps.screening.application.specification.impl.DefaulSubmissionValidatorFactory:

1. validators – required parameter, multi-value; defines the list of validator names to use;
2. <validator_name>_class – required parameters; define the fully qualified class name of the *SubmissionValidator* to use;
3. <validator_name>_namespace – optional parameters; define the ConfigManager namespace for the validator. If absent, the default constructor will be used to create the validator.
4. formatter_class – required parameter; defines the fully qualified class name of the *ValidationOutputFormatter* to use;
5. formatter_namespace – optional parameter; defines the ConfigManager namespace for the formatter. If absent, the default constructor will be used to create the formatter.

3.3Dependencies Configuration

None

4.Usage Notes

4.1Required steps to test the component

1. Extract the component distribution.
2. Execute 'ant test' within the directory that the distribution was extracted to.

4.2Required steps to use the component

None

4.3Demo

The demo assumes that the default component configuration is used. This configuration is given below.

```
<?xml version="1.0" encoding="UTF-8"?>

<ConfigManager>

<namespace
name="com.topcoder.apps.screening.application.specification.ValidationManager">

       <Property name="factory_class">

              <Value>

              com.topcoder.apps.screening.application.specification.impl.Default
              SubmissionValidatorFactory

              </Value>

       </Property>

</namespace>
```

```
<namespace
name="com.topcoder.apps.screening.application.specification.impl.DefaultSubmissi
onValidatorFactory">
	<Property name="formatter_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.formatt
		ers.XMLValidationOutputFormatter
		</Value>
	</Property>
	<Property name="validators">
		<Value>ucd_validator</Value>
		<Value>ucd_naming_validator</Value>
		<Value>ad_validator</Value>
		<Value>ad_naming_validator</Value>
		<Value>ad_report_generator</Value>
	</Property>
	<Property name="ucd_validator_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.validat
		ors.DefaultUseCaseDiagramValidator
		</Value>
	</Property>
	<Property name="ucd_naming_validator_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.validat
		ors.DefaultUseCaseDiagramNamingValidator
		</Value>
	</Property>
	<Property name="ad_validator_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.validat
		ors.DefaultActivityDiagramValidator
		</Value>
	</Property>
	<Property name="ad_naming_validator_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.validat
		ors.DefaultActivityDiagramNamingValidator
		</Value>
	</Property>
	<Property name="ad_report_generator_class">
		<Value>
		com.topcoder.apps.screening.application.specification.impl.validat
		ors.ActivityDiagramPathReportGenerator
		</Value>
	</Property>
```

```
</namespace>
</ConfigManager>
```

Assume that the sample XMI file provided with the design distribution ("TerrierCorpEComSpec60813.xmi") is used.
After the component was configured appropriately, validation of the file and generation of an XML output of all unique activity diagram paths requires just several lines of code.

```
// create an instance of the ValidationManager
// we use the default constructor which loads the configuration
// from the default namespace (fully qualified class name)
ValidationManager m = null;

try {
    m = new ValidationManager();
} catch (ConfigurationException e) {
    // something happened when loading the configuration
    …
} catch (XMIParserException e) {
    // failed to parse the XMI file
    …
}

// perform the validation
String[] output = null;

try {
    output = m.validate(new File(xmiFileName));
} catch (FormatterException e) {
    // failed to format the output
    // shouldn't happen with the default configuration
    …
}

// print the output data to the console
for(int i=0; i < output.length; i++)
    System.out.println(output[i]);
```

Full output for the sample file is too huge to be shown here. The fragment below represents a report (without any error information) generated for a single activity diagram named "Edit Product in Catalog Activity Diagram". The report will contain two paths. Note that there will be no errors in the output data because the sample XMI file provided is fully valid according to the default TC guidelines.

```
<validation>
<errors>
</errors>
<reports>
<diagram id="I1c882dbm10485cc4cacmm75f4" type="ActivityDiagram"
name="Edit Product In Catalog Activity Diagram”>
<report>
<Path number=”1”>
<Node>Start</Node>
<Node>Display Product List</Node>
<Node>User Selects Product</Node>
<Node>Display Current Product Information</Node>
<Node>User Updates Information</Node>
<Node>Validate Information – information valid</Node>
</report>
<report>
<Path number=”2”>
<Node>Start</Node>
<Node>Display Product List</Node>
<Node>User Selects Product</Node>
<Node>Display Current Product Information</Node>
```

```
<Node>User Updates Information</Node>
<Node>Validate Information – else</Node>
<Node>User Updates Information</Node>
<Node>Validate Information – information valid</Node>
</report>
</diagram>
…
</validation>
```

Application developers can provide custom validators and formatters. For example, assume that in the previous example the following custom formatter was used instead of the default one.

```
// custom formatter implementation
public class CustomFormatter implements ValidationOutputFormatter
{
     public CustomFormatter() {}
     public String[] format(ValidationOutput[] output)
     {
          String[] ret = new String[1];

          int countErrors = 0:
          for (int i = 0; i < output.length; i++)
          {
              if (output[i].getType() == ValidationOutputType.ERROR)
              {
                  countErrors = 0;
              }
          }

          if (countErrors == 0)
          {
              ret[0] = "The submission is valid.";
              ret[1] = "Errors found: 0.";
          }
          else
          {
              ret[0] = "The submission is invalid.";
              ret[1] = "Errors found: " + Integer.valueOf(countErrors)
                       + ".";
          }

          return ret;
     }
}
```

The formatted output for sample submission would be

```
The submission is valid.
Errors found: 0.
```

5.Future Directions

- additional validators and report generators can be defined;
- additional formatter for validation output can be defined.