Base Exception 2.0 Component Specification

1. Design

The Base Exception 2.0 component builds on the central idea to the version 1.0 component - to provide a family of exception classes that can be extended later by custom exceptions, allowing the exceptions to contain more information than Java built-in exceptions.

Each exception / error provided allows much more information to be attached to it, including:

- Application, Module and Error codes, to allow for simple identification of the cause of an error within an application. These are set on construction and cannot be changed afterwards.
- Runtime information, currently thread name and creation date. These help describe the environment that the exception was raised by.
- Logging flag - while this component does not provide logging, it does give access to a flag indicating whether the exception has been logged
- Key/Value information - independent to those listed above, a user can attach any [key, value] pairs to the exceptions, which can then be read later up the handling chain. This also allows the greater customization of exceptions in the future.

The data is set on construction of the exception, and can then be accessed only by calling methods on the exception itself - setXXX methods return the data instance, to allow only the required values to be set easily, similar to how the named-parameter pattern works (e.g. setInformation("name", "bob").setErrorCode("a112").setLogged(true)).

Base classes for the three major types of exceptions are provided - BaseRuntimeException and BaseError exception are updated from version 1.0 to allow more information to be added. BaseException has been provided, but is now conceptually split into two lines:

- "Critical" exceptions, for problems that cannot be reasonably handled by an application - these may require the application to terminate, or human intervention.
- "Non-critical" exceptions, for problems which can be handled by the application, without the need for any external action.

While currently these two classes are nearly identical, the split has been provided in the event that later they will have different logic applied to each.

Finally, to assist in the use of localized messages, a utility class ExceptionUtils has been provided to retrieve error messages from a resource bundle. Along with this, common exception-raising checks are provided to reduce the amount of code rewritten for any developer using these exceptions in their application.

1.1 Design Patterns

Delegate: BaseException, BaseRuntimeException and BaseError all store their data within a separate ExceptionData instance, and all get/set of data thereafter is delegated to this member.

1.2 Industry Standards

None

1.3 Required Algorithms

None

1.4 Component Class Overview

ExceptionData:

Container class for all data associated to an exception within this component - such as error/module codes, creation information, a logging flag and unrestricted key/value pairs. When first created, each data instance is filled with default values, and the thread name and creation date are set to their current values. The remaining members all have setXXX methods, which all return the 'this' instance to allow the data to be populated on a single line. Each member also has a getXXX method for access.

ExceptionUtils:

Utility class that provides static access to actions performed frequently when dealing with exceptions. This includes a way to retrieve a localized message using a resource bundle, and two common parameter checking methods (for null objects or empty strings), which also utilize messages from resource bundles. It is the intention that any popular check methods within TCS components can go within this class, so they are not required to be rewritten for each component. All members are serializable, except the map if it contains key or value objects which cannot be serialized - these should be avoided if serialization is required.

CauseUtils:

Utility Class maintained to provide backwards compatibility with Base Exception 1.0. As this component is to be used in an environment with automatic exception chaining (java 1.4 and later) this is not required, but for compatibility it has been provided, its single method rewritten to match the new technology.

1.5 Component Exception Definitions

NB - all exceptions below are defined by this component, not thrown by it. In fact, it is recommended (but not required) that custom exceptions extending these do not throw exceptions from their methods.

BaseException:

BaseException is the eventual superclass for any custom exceptions used within TC software components - it provides all the logic of Java 1.4 exceptions (message, stack trace,..) but also includes a collection of additional information about the exception, such as module and error codes for identification, a transient logged flag, trigger date/time/thread as well as unlimited key/value pairs.

BaseCriticalException:

This exception is the superclass for any TCS exceptions that are considered 'critical'. This is to be extended by exceptions which cannot be realistically handled within an application - for example, if a database dies or network connection can't be established. These may require the application to halt and human intervention to take place. Currently, it just passes each constructor's parameters to its superclass, but has been split from BaseException to allow its logic to be different from BaseNonCriticalException in the future.

BaseNonCriticalException:

This exception is the superclass for any TCS exceptions that are considered 'non-critical'. This is to be extended by exceptions which can be realistically handled within an application - for example, if a user enters invalid data, or an optional resource cannot be located. Currently, it just passes each constructor's

parameters to its superclass, but has been split from BaseException to allow its logic to be different from BaseCriticalException in the future.

BaseRuntimeException:

BaseRuntimeException is the eventual superclass for any custom runtime exceptions used within TC software components - it provides all the logic of Java 1.4 exceptions (message, stack trace,..) but also includes a collection of additional information about the exception, such as module and error codes for identification, a transient logged flag, trigger date/time/thread as well as unlimited key/value pairs. This exception is similar to BaseNonCriticalException, except methods are not required to declare that they throw BaseRuntimeExceptions.

BaseError:

BaseError is the eventual superclass for any custom Errors used within TC software components - it provides all the logic of Java 1.4 exceptions (message, stack trace,..) but also includes a collection of additional information about the exception, such as module and error codes for identification, a transient logged flag, trigger date/time/thread as well as unlimited key/value pairs. This exception is similar to BaseCriticalException, except that Errors should represent abnormal problems that should not occur, and don't necessarily have to be declared as thrown or handled.

1.6 Thread Safety

The exceptions provided within this component are not thread safe, they allow their internal data to be modified and retrieved by multiple threads with no protection. It could be the case that one thread attempts to modify an information object while another is reading it. However, this should be of little consequence as the intended behavior of the exceptions is not to be handled by multiple threads. Additionally, the two utility classes are stateless, and so thread safe.

2. Environment Requirements

2.1 Environment

- Java 1.4 for development, 1.4 and 1.5 for compilation.

2.2 TopCoder Software Components

- None

2.3 Third Party Components

- None

3. Installation and Configuration

3.1 Package Name

`com.topcoder.util.errorhandling`

3.2 Configuration Parameters

None required, though a resource bundle will have to be created if localization is desired.

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

BaseException is used to provide exceptions that are extended by custom exceptions within other components. To do this, one must choose which base class to extend:

Handled:	**Checked:**	**Unchecked:**
Within application	BaseNonCriticalException	BaseRuntimeException
Outside application	BaseCriticalException	BaseError

Once the superclass is chosen, the required constructors should be implemented to call the constructors on the superclass, adding the module and error data if desired.

4.3 Demo

4.3.1 Create Base Exceptions

// Creating base exceptions, calling their constructors is enough. Since the constructors are
// similar, we focus on Base Exception

```
    // create ExceptionData instance used to construct exceptions
    ExceptionData data = new ExceptionData();
    // create Throwable instance used to construct exceptions
    Throwable throwable = new NullPointerException();
    // create a BaseException instance with its constructors
    BaseException baseException = new BaseException();
    baseException = new BaseException("test");
   baseException = new BaseException(throwable);
   baseException = new BaseException(data);
   baseException = new BaseException("test",throwable);
   baseException = new BaseException("test",data);
   baseException = new BaseException(throwable,data);
   baseException = new BaseException("test",throwable,data);

    BaseNonCriticalException nonCriticalException = new
                                BaseNonCriticalException();
    ...  // construct with other BaseNonCriticalException constructors
    BaseCriticalException criticalException = new BaseCriticalException();
    ...  // construct with other BaseCriticalException constructors
    BaseError error = new BaseError();
    ...  // construct with other BaseError constructors
    BaseRuntimeException runtimeException = new BaseRuntimeException();
    ...  // construct with other BaseRuntimeException constructors
```

4.3.2 Write a custom exception

// In this section, we demonstrate how to write a custom exception. The defined exception will be
// used in the following sections:

```
public class CustomException extends BaseNonCriticalException {
    public static final String APPCODE = "UserHandler"; // application code
    public static final String MODCODE = "MY_MODULE";   // module code
    public static final String ERRCODE = "e1421_m";     // error code

    public CustomException(String message){
        this(message, null, new ExceptionData());  // call generic constructor
    }

    public CustomException(String message, Throwable cause){
        this(message, cause, new ExceptionData()); // call generic constructor
        }
```

```
    public CustomException(String message, Throwable cause, ExceptionData data){

        // call superconstructor after setting the module and error codes
        super(message, cause, getData(data);
    }

  // get ExceptionData instance: if passed data is null, a new ExceptionData
  // is used instead
    private static ExceptionData getData(ExceptionData data) {
        if (data == null) {
              return new ExceptionData();
        } else {
              Return data.setApplicationCode(APPCODE).setModuleCode(MODCODE)
                          .setErrorCode(ERRCODE);
        }
    }
}
```

4.3.3 Use the exception within an application

// In this section, we demonstrate how to use the exception within an application

Suppose the following ResourceBundle class is defined in the same package (com.topcoder.util.errorhandling) :

```
public class MyResourceBundle extends ListResourceBundle {

    private static final Object[][] contents = {
        {"key", "value"}, {"doubleKey", new Double(0.0)}, {"emptyKey", ""},
        {"trimemptyKey", "  "}, {"sameKey","sameKey"},
        {"app-module-error","value1"}, {“badname”, “bad”},
        {“nospace”, “space space”}, {“appCode-SmartEx-e617”, “message”}
    };

    public Object[][] getContents() {
        return contents;
    }
}
```

Then, please refer to CS 4.3.2 for the details of CustomException. Following is a demonstration of using the exception within an application:

```
public void throwMethod(String name)
        throws CustomException {

    // Check for valid name, throws IAE if string is empty or null.
    ExceptionUtils.checkNullOrEmpty(name, null, null, “No empty name”);
    // obtain resource bundle
    ResourceBundle bund = ResourceBundle.getBundle(
          “com.topcoder.util.errorhandling.MyResourceBundle”);

    // use simple constructor, similar to Base Exception 1.0
    if(name.equals(“baduser”)){
        throw new CustomException(
            ExceptionUtils.getMessage(bund, “badname”, “Bad user!”));
    }

       // use data constructor to attach more information
    if(name.indexOf(“ “) != -1){
        System.out.println(“Encounter a name containing blank spaces”);
                                              // assume it is logged somehow
        throw new CustomException(
            ExceptionUtils.getMessage(bund, “nospace”, “No spaces”),
            null, // no ‘cause’ for this exception,
```

```
            new ExceptionData().setInformation(“attempted name”, name)
                               .setInformation(“size”, new Long(name.length()))
                               .setLogged(true)); // set the exception data
    }
}

// ... somewhere else in the application:
public void catchMethod() {
    try{
        throwMethod(“baduser”);
    } catch(CustomException e){
        // log the exception if not already:
        if(e.isLogged() == false){
            e.setLogged(true);
            System.out.println(e.getErrorCode() + “ from ” +e.getModuleCode());
            // prints ‘e1421_m from MY_MODULE’
        }
    }

    try{
        throwMethod(“spaced name”);
    } catch(CustomException ce){
        System.out.println(“Logged on thread ” + ce.getThreadName());
        System.out.println(“Logged at time ” + ce.getCreationDate());

        String bad = (String)ce.getInformation(“attempted name”);
        if(bad != null){
           // perform some other logic on the exception
           ce.setInformation(“suggested name”, bad.replaceAll(“ “, “_”));
        }
    }
}
```

4.3.4 Write a custom exception making use of the given utilities to provide exception message

// In this section, we demonstrate how to write a custom exception which makes use of the given
// utilities to provide exception message.

Suppose the following AppUtils class is defined in the same package:

```
public class AppUtils {
    public static String getAppCode() {
        return "appCode";
    }

    public static ResourceBundle getResourceBundle() {
        return ResourceBundle.getBundle(
                   "com.topcoder.util.errorhandling.MyResourceBundle");
    }
}
```

Following is a custom exception — SmartException:

```
public class SmartException extends BaseCriticalException {
    public static final String MODCODE = “SmartEx”; // module code

    // Build exception from error code, cause, data and logger
    public SmartException(String error, Throwable cause,
                          ExceptionData data, Log logger){

        // call super constructor, statically building all information
        super( buildMessage(error), cause, getData(data, error));

        // now log if needed
```

```
        if(logger != null){
            logger.log(Level.ERROR, getMessage());
            setLogged(true);
        }
    }

    // get ExceptionData instance: if passed data is null, a new ExceptionData
    // is used instead
    private static ExceptionData getData(ExceptionData data, String error) {
        if (data == null) {
              return new ExceptionData();
        } else {
              return data.setApplicationCode(AppUtils.getAppCode())
                         .setModuleCode(MODCODE)
                         .setErrorCode(error);
        }
    }


    // static automated message building
    private static String buildMessage(String errCode){
        // get application data (from fictitious static AppUtils)
        String         appCode = AppUtils.getAppCode();
        ResourceBundle bundle  = AppUtils.getResourceBundle();

        // parse statically from utilities
        return ExceptionUtils.getMessage(bundle,  appCode,
                                         MODCODE, errCode);
        // should start appCode + "-SmartEx-" + errCode,
        //   with optional error message prefix if found in bundle.
    }
}

// later, SmartException can be thrown somewhere
    throw new SmartException("e617", null, new ExceptionData(), myLogger);
```

4.3.5 Usage of class ExceptionUtils

// In this section, we demonstrate how to use utility class ExceptionUtils

```
    // obtain resource bundle
    ResourceBundle bund = ResourceBundle.getBundle(
              "com.topcoder.util.errorhandling.MyResourceBundle");
    // get localized message string
    System.out.println(ExceptionUtils.getMessage(bund,"key","default"));
    System.out.println(ExceptionUtils.getMessage(null,null,"default"));
    // get an exception message automatically from the exception's codes
    System.out.println(ExceptionUtils.getMessage(bund,"app","module","error"));
    // check null argument, IllegalArgumentException is thrown
    try {
       ExceptionUtils.checkNull(null,null,null,"null value");
       fail ("IllegalArgumentException expected.");
    } catch(IllegalArgumentException iae) {
       // good
    }
    // check empty String, IllegalArgumentException is thrown
    try {
       ExceptionUtils.checkNullOrEmpty(" ",null,null,"empty value");
       fail ("IllegalArgumentException expected.");
    } catch(IllegalArgumentException iae) {
       // good
    }
```

5. Future Enhancements

One possible enhancement would be to provide even more methods inside the ExceptionUtils class, to help avoid each TCS developer having to add similar methods to each component separately.

Another may be to provide some stack trace analysis, for example giving the exceptions methods like .getThrowingClass / getThrowingMethod / getThrowingLine etc.