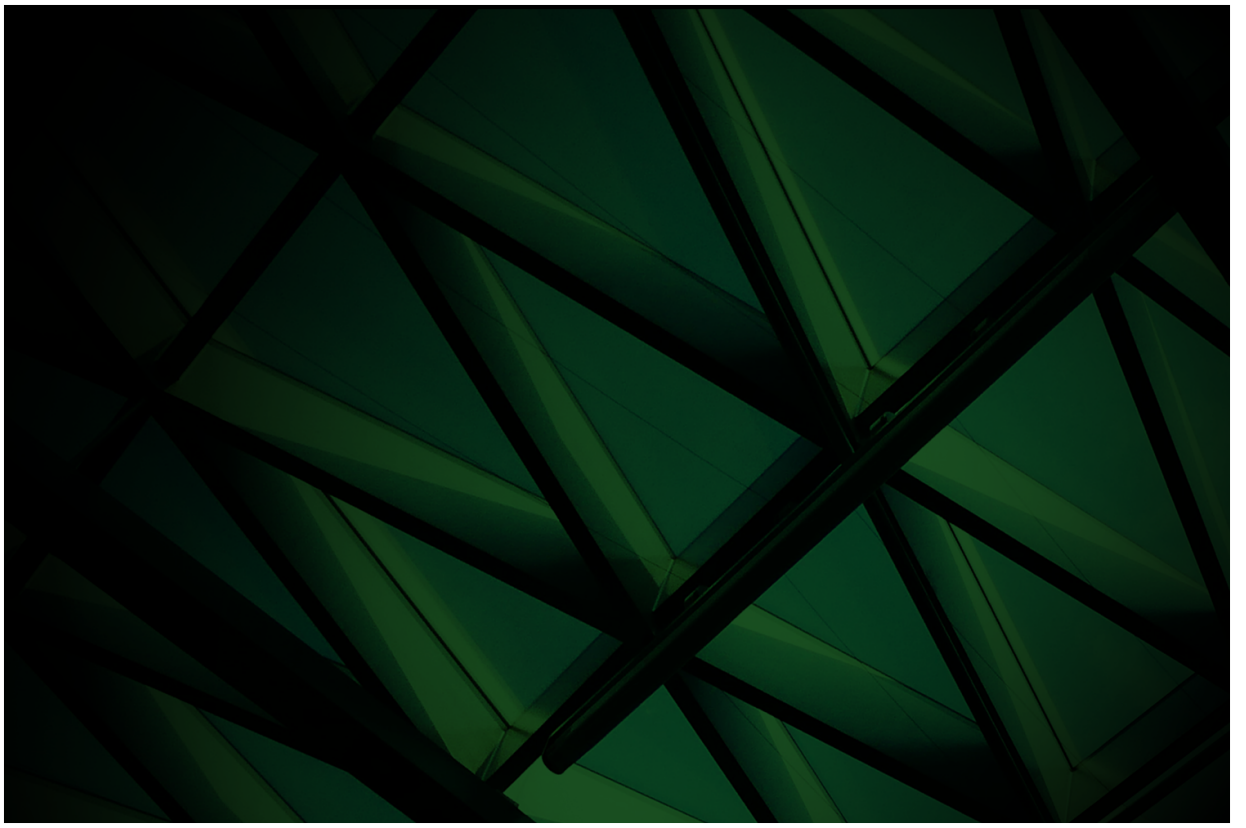

Logical Certifier White Paper

VERSION 2022

jade platform

Contents

Logical Certifier

The Jade Logical Certifier is a tool (application) that is used to verify and, if necessary, restore the referential integrity in a Jade database.

This white paper discusses the Jade Logical Certifier and its usage; that is:

- Why the Jade Logical Certifier is needed
- What the Jade Logical Certifier does
- How to run the Jade Logical Certifier
- How to interpret and repair the errors the Jade Logical Certifier produces
- How often you should run the Jade Logical Certifier
- How to restrict analysis coverage

This document uses an example that describes a simple domain model (managing a hardware company's products) from which you can create a corresponding simple schema with some database objects.

Exercises towards the end of this document demonstrate some examples of broken referential integrity in this simple schema and how you can use the Jade Logical Certifier to restore referential integrity.

For more details about the Jade Logical Certifier, see the following subsections.

Referential Integrity

A Jade inverse reference defines a contract specifying the knowledge that two types of entities have about each other. Jade supports cardinalities one-to-one, one-to-many, and many-to-many.

In Jade, the relationships are managed using object identifiers (oids); for example, consider the following one-to-many relationship.

```
Company <──> Product
```

Company::**allMyProducts** is an automatic inverse reference of **Product**::**myCompany**. If a **Product** instance **P** exists that references a **Company** instance **C**, the contract requires that **P** is a member of **C**::**allMyProducts**. The absence of this member constitutes a lack of referential integrity and would be detected by the Jade Logical Certifier as an error (identified as error 1 in "Logical Certifier Errors and Repairs", in Chapter 5 of the *Object Manager Guide*).

Conversely, if **P** is a member of **C**::**allMyProducts**, **P**::**myCompany** must reference **C**.

Possible Issues with Referential Integrity

Jade should maintain referential integrity when a relationship is set up with a Jade-defined inverse. In the rare event that such an inverse was not maintained correctly, that could be indicative of a Jade fault. If it can be shown that an error in user data occurred in a recent Jade release and it is not a historical issue, if your Jade licenses include support, contact your local Jade support center or Jade Support.

Sometimes data is lost through corruption in database storage media. In such cases, it is expected that a backup should be recoverable from a separate storage medium. In the event that a correct backup is not available, running the Logical Certifier to restore referential integrity is usually part of the repair procedure.

Consequences of Referential Integrity Problems

Referential integrity problems can result in data inaccessibility, the wrong data being accessed, or runtime exceptions being generated.

The business impact of a referential integrity problem is not determined by the classification of that problem (as identified in "Logical Certifier Errors and Repairs", in Chapter 5 of the *Object Manager Guide*). Business impact is determined by how the anomaly impacts upon workflows associated with the entities concerned.

The Jade Logical Certifier and Referential Integrity

The Logical Certifier is run in two modes.

- In analysis mode, the Logical Certifier checks referential integrity, records errors found, and generates corresponding fixes.
- In repair mode, the Logical Certifier performs the fixes generated in the prior analysis.

The logical certification process analyzes each class in a schema and performs various cross-checks on each instance, to verify its logical/referential integrity.

The term *certify* is ambiguous, since it is an incomplete term used in the context of both the Jade Logical Certifier (performing a *logical certify*) and also in the context of the Jade Database utility (performing a *file certify* via the **Certify Files** operation that checks on the physical structure and format of **.dat** files). To avoid confusion, it is preferable to use the complete term.

Manually Maintained References

The Logical Certifier does not check for errors in cases when there is no inverse defined. It is your responsibility to ensure referential integrity for manually maintained collections (not managed via a Jade inverse relationship).

A manually maintained collection can contain invalid object references or invalid keys, and these will not be reported by the Logical Certifier.

Examples of User Data Errors and Fixes

There are a number errors for which the Logical Certifier checks. This document briefly discusses three of these. See "Logical Certifier Errors and Repairs", in Chapter 5 of the *Object Manager Guide*), for a complete list of the checks. Consider again the case of the following inverse relationship.

```
Company::allMyProducts <──> Product::myCompany
```

In this inverse relationship:

- If a product (for example, **"box of nails"**) references its company but the product is not held in the collection of products known to the company, this would be reported by the Logical Certifier as an Error 1 *(Missing Reference to a Collection)*.

  The fix provided by the Logical Certifier is to insert the product into the collection by performing a collection rebuild. This scenario is the subject of Exercise Two.

- If a product has since been removed from stock (the product instance has been removed from the database) but the product was not removed from the company's collection of products, the reference to the product in the collection is invalid.

This circumstance would be reported by the Logical Certifier Error 12 *(Collection contains invalid reference)*. The fix provided by the Logical Certifier is to remove the invalid product reference from the collection by performing a collection rebuild.

- If a product (for example, **"box of nails"**) is included in the company's collection of products but is stored at the wrong key (for example, **"carton of nails"**), the product would not be found in the collection using the member's key property value. This might occur if the product key (**product::description**) was changed; however, the collection was not updated correctly. The Logical Certifier will identify two errors in this scenario.

  When analyzed from the **Company** side of the inverse, the Logical Certifier will report Error 3 *(Collection contains object at invalid keys)*. The fix provided by the Logical Certifier is to move the product in the collection and correct the key by performing a collection rebuild. When analyzed from the **Product** side, the Logical Certifier will report Error 1 *(Collection does not include reference to inverse object)*, the same error as reported in example 1, above. This is because the Logical Certifier implicitly does a lookup in the dictionary using the actual key property value (using the description **"carton of nails"**). The nature of this fix depends on the update mode for the inverse; this aspect of how the Logical Certifier works is discussed in "Effect of Update Mode", later in this document. The fix for this error will not be run, however, because the execution of the fix is conditional upon the circumstance still existing. The rebuild fix generated for the Error 3 will be run first, meaning that the product will then be found so that the second fix will be skipped. This scenario is the subject of Exercise Six, part D, later in this document.

Tip If you want to perform an online repair of a collection while it is concurrently updated, call the **repairCollection** method of the **Application** class, to remove invalid object references and fix up dictionary keys in a specified collection. This is sometimes faster than a Logical Certifier rebuild repair.

For more details, see "Chapter 5 – The Jade Logical Certifier Diagnostic Utility", in the *Object Manager Guide*.

User Data Metadata Distinction

User data objects are instances of user classes defined by you. The example in the previous section describes using the Logical Certifier to analyze the referential integrity of such objects (that is, **Company** and **Product**).

The Logical Certifier includes support for analyzing the logical integrity of the metamodel. The metamodel consists of Jade system classes that describe the user model (user schemas). The term *metamodel* is appropriate, because it is a self-describing system; Jade system classes are used to describe classes (and class methods and properties). The metamodel consists of instances of classes such as **Class**, **ExplicitInverseRef**, **Inverse**, **JadeMethod**, **Parameter**, **ReturnType**, and so on. These metamodel entities are created, modified, and deleted during development or during a schema deployment. In Exercise One, a schema is created that represents the relationship between a company and its products, and some of the metamodel elements used to represent this model are inspected.

Metadata errors can have application runtime consequences, but typically result when making changes to the user model, either via the Jade Platform development environment or during a schema load; for example, an error 4 *(Object not found)* or 1090 *(Attempted access via null object reference)* may be raised when performing a reorganization). Often the role of the Logical Certifier in such cases is to analyze the referential integrity of the metamodel prior to the model change, and repair the errors so that the required model changes can then be implemented.

Performing a logical certify of metadata can take a long time to complete. Similarly, performing a logical certify of user data can take a long time to complete, due to the number of objects that will be analyzed. For this reason, analysis coverage can be restricted by schema or class (for details, see "Restricting the Analysis Coverage", later in this document).

When metadata is analyzed, the opening line of the **_logcert.log** file will have an entry of the form:

```
21 September 11:53:00: Analysis of user schemas meta-structure started
```

This provides a means by which you can quickly verify that you have initiated the intended function. You can abort the operation if you selected the wrong function.

Metamodel Example Errors

This section contains examples of metadata errors.

Caution If your Jade licenses include support, contact your local Jade support center or Jade Support before attempting metaschema repairs.

- An object that has a reference (manual) to its parent but the object is missing from the inverse collection on the parent.

  The add action provided by the Logical Certifier repairs the anomaly.

  ```
  *** Error 1: Collection does not include reference to inverse object
  JadeUserProfile/1749.135->UserProfile::typesChanged (inverse of
  GUIClass/1284.1155)
  FIX40: add 1429.135.1749.2.4 1284.1155
  ```

- A collection has an invalid member type, which the Logical Certifier will report as:

  ```
  *** Error 2: CollClass::memberType references invalid object.
  ```

  The appropriate action to take depends on other circumstances reported by the Logical Certifier, which is an example of why it is a good idea to contact your local Jade support center or Jade Support for advice if your Jade licenses include support.

Running the Logical Certifier

You can run the Logical Certifier from a GUI application (accessed from the **Jade Logical Certifier** shortcut in the Jade Utilities submenu of your Jade installation in the Start menu). Alternatively, you can run the Logical Certifier as a RootSchema application in a single-user development session, by clicking the **Run Application** toolbar button or in batch mode. In batch mode, which has the following format, the **operation** argument specifies the mode for the invocation.

```
jadclient path=database-path ini=Jade-initialization-file-path schema=RootSchema
app=JadeLogicalCertifierNonGui server=SingleUser startAppParameters
operation=certify|certifyMeta|repair logDir=log-file-disk-path
```

The **logDir** parameter specifies the path to which the various log files are created (for details, see the following section). It is also the location in which a **_logcert.in** file is placed when running an analysis of a subset of the database (for details, see "Restricting the Analysis Coverage", later in this document).

Tasks Performed by the Logical Certifier

The Logical Certifier examines metadata to determine classes and their relationships, and builds a transient method that performs all of the integrity checks when executed.

The analysis results in the following four files being output to the specified directory.

- **_logcert.err**, which provides a description of the errors found and gives repairs
- **_logcert.fix**, which contains the fixes executed when the Logical Certifier is run in repair mode
- **_logcert.log**, which provides a running commentary of the analysis
- **_logcert.cls**, which provides a list of classes for which errors were detected

When the repair is run, the following files are created.

- **_repair.log**, which provides a commentary of the repair process
- **_repair.err**, which lists repairs that were unable to be actioned (normally none)

In addition, the fix file is renamed **_logcertBackup.fix**.

Although many errors can be repaired without your input, some situations require you to decide between alternative repairs (for details, see Exercise Six, Part C). In addition, situations involving duplicate keys require you to write a script to repair the error. This is discussed in the section "Analysis and Repair Process". (See also Exercise Six, Part E.)

Precautions

It is important that no reorganization of the database has been performed after the analysis, as the reorganization may invalidate the repairs.

The Logical Certifier does not take into account the significance of all combinations of errors, nor the significance of the types involved. The fixes provided by the analysis process are prescriptive and are derived from the classification of the error type (documented in "Logical Certifier Errors and Repairs", in Chapter 5 of the *Object Manager Guide*).

The Logical Certifier also applies rules that take into account the update mode (manual, automatic, or man/auto) in determining what fix is reasonable.

For these reasons, if your Jade licenses include support, contact your local Jade support center or Jade Support before attempting metadata repairs.

Often when providing files documenting analysis of metadata to Jade Support, it is useful to also provide the files constituting the system description (that is, **_user*.dat** and **_control.dat**) so that the model can be examined first-hand.

As with any database-affecting operation, a sound backup and change management process is required.

Backup Strategy

A full user data analysis is usually performed on a recent backup copy of the database.

For the repair, the following strategies are recommended.

- Run the repair on a backup copy of the system and test the result before running the repair against the production system.

  You should still have a backup copy that you can restore.

- Take a backup immediately prior to running the repair against the production system (the backup should be verified).

  Test the result and restore from backup, if necessary.

When to Run the Logical Certifier

This section describes when to run the Logical Certifier diagnostic utility.

- When to Run the Logical Certifier to Analyze Metadata
- When to Run the Logical Certifier to Analyze User Data

When to Run the Logical Certifier to Analyze Metadata

In a production system, metadata would typically be analyzed following a deployment of schema changes.

In development systems, metadata should be analyzed regularly, especially during the development phase in which the model changes are frequent.

When to Run the Logical Certifier to Analyze User Data

In production systems, user data should be analyzed regularly, since this data is volatile and affects the end-user's business.

In development systems, user data would typically be analyzed regularly, to ensure the validity of the testing process.

Restricting the Analysis Coverage

It is sometimes useful to perform an analysis on a subset of the model. The restriction is defined through selections you make in the user interface if you are running the GUI Logical Certifier or via specifications in a **_logcert.in** file if you are running the Logical Certifier in batch mode. Examples of situations requiring this are when:

- A runtime anomaly suggests a problem in a specific area.
- Performing a follow-up analysis after a repair.

  You can rename the **_logcert.cls** file produced by the prior analysis that lists classes for which errors were detected to **_logcert.in**, which is read by the subsequent analysis.
- Checking an area of the model that is known to have encountered errors in the past.

Typically, the analysis coverage would be limited by specifying a subset of classes, but you can restrict the analysis by specifying:

- An instance range
- An instance date-created range
- Inclusion or exclusion of subclasses

For a meta-certify operation, you can use the **_metacert.in** file to specify the **RootSchema** classes that are to be certified. The format of the **_metacert.in** file is the same as that of the **_logcert.in** file except that the **Schema** and **AllSchemas** values are not valid, because the certification of meta data is performed at **RootSchema** level so you cannot filter the schema selection.

For more details, see "Chapter 5 – The Jade Logical Certifier Diagnostic Utility", in the *Object Manager Guide*.

Note It is important to run a full logical certify and repair on occasion, so that known clean points are established. This is important because fixing the errors promptly minimizes potential runtime negative consequences. In addition, knowing the version of Jade in which an error was induced is relevant to determining whether the error is the result of a known cause that has been addressed.

A reproducer involves taking a database from a known clean state to an erroneous state through a set of well-defined steps. A Logical Certifier analysis is required to verify the start and end states.

Analysis and Repair Process

This document earlier described the analysis and repair modes in which you can run the Logical Certifier. In practice, some decision-making is sometimes involved and a process that is iterative and that includes manual checking of the generated logs is recommended.

The following sections provide some analysis and repair scenarios and suggest a general process that is appropriate.

Effect of Update Mode

As the "Precautions" section earlier in this document stated, when two entities are inconsistent, the following question arises.

```
Which entity is correct and which needs amendment?
```

The Logical Certifier decides which fix is appropriate, by treating the manual side of an inverse reference as being correct.

In Exercise Six, Part A, the scenario is provided in which a product does not know its company but the company's collection of products includes this product. When the Logical Certifier analyzes this scenario, the error reported and the fix generated depend on the update mode of the following relationship.

```
Product::myCompany <—> Company::allMyProducts
```

If the reference **Product::myCompany** is defined as manual (and the collection is automatically updated), the assumption made by the Logical Certifier is that the reference was intentionally set to null and for some reason inverse maintenance failed, so the product was not removed from the company's collection of products. In this case, the repair action is to remove the product from **Company::allMyProducts**.

However, if the collection is manually updated and the reference on **Product** is automatically set, the fix generated by the Logical Certifier is to set the reference **Product::myCompany** (based on the assumption that the product was intentionally added to the collection of products referenced by the company).

If the inverse is defined as man/auto on both sides, the rule of assuming the manual side to be correct does not apply. In such cases, you will be required to edit the repair file to select the correct repair option presented (when options are commented out). This scenario is demonstrated in Exercise Six, Part C. Which of these two repairs is correct depends on the model and requires your decision. You may know, for example, that even though the inverse is defined as man/auto, the update is always performed from one side, so that side should be regarded as the correct reference.

Tip Examine the error file before the repair is run, to see if you need to make fix choices.

A Repair that is Not Actioned

A repair may not be actioned because it violates some other principle. For example, the repair may result in a duplicate key for a dictionary that is defined as not allowing duplicates.

The **_repair.err** file reports the fact that the fix was not run as a result of the condition encountered, as follows.

```
Error: FIX1 rebuild AllProductsByDescription/2976.1.2973.2.1: size before=2, size
after=2 results in duplicate
```

Tip After a repair is run, check the **_repair.err** log for the details of any fixes that were not run, before you decide what action to take next.

This scenario is the subject of Exercise Six, Part E, later in this document.

Fixes are Evaluated Independently

As described earlier in this document, the Logical Certifier iterates through instances of each class and performs various checks after looking at what properties are defined on the class that have inverses. This means that for a specific inverse, checks are performed on instances of both classes participating in the relationship, with the result that errors reported often occur in pairs (one for each side of the inverse).

When finding an error, the Logical Certifier does not link this to an error discovered earlier while investigating the relationship from the other side. Each repair does, however, come with a check clause that specifies the condition that gave rise to the fix. If this condition no longer applies, the fix is not run (perhaps because the fix applied as the result of analyzing the other side of the relationship resolved the issue).

When an object is stored at a key different from its key property value, two errors are reported. An error that the member is not stored at the correct key is reported from the collection owner side. Another error that the member cannot be found in the collection on the other side is reported from the member side. It is not known whether the object reference is absent or if it is stored at the wrong key. The fix generated for this depends on the update mode of the inverse. When the rebuild fails (as described earlier in this document), the final state is determined by the fix that the Logical Certifier applies in accordance with the update mode.

Because you should perform this investigation on a backup copy of the system, typically you would decide to write a script that deals with the key property value that results in a duplicate key. (It may be possible to identify the appropriate unique value from other identifiers on the member, and set the correct key value.)

Tip Evaluate what other changes may have occurred as the result of a fix not running. Typically the last action you perform is not a repair; it is a final Logical Certifier analysis that confirms that there are no more errors.

This scenario is the subject of Exercise Six, Part E, later in this document.

Summary of Analysis and Repair Process

When considering the scenarios described in earlier sections of this document, it is advisable to analyze the files created by the Logical Certifier after each invocation before deciding what to do next.

After each analysis phase, check for fixes that require a choice to be made, and also consider whether any scripted solutions are appropriate.

After each repair, check the repair log and repair error log for fixes that were not executed, and evaluate the cause of this while also giving consideration to how this affects the end state.

An iterative process that terminates with an error-free analysis is recommended.

Practical Exercises

The following subsections contain practical exercises, which are provided so that you can step through the process of analyzing and repairing a small system. Part of this process involves using the Logical Certifier with a crafted fix file, to intentionally break referential integrity.

Before You Begin

We recommend that you use a throw-away system for these exercises.

You can decide to extract the schema following the completion of Exercise One. This will enable you to establish a clean initial state by removing the schema, loading the schema extract, and then running the **JadeScript::setup** method from Exercise One, to recreate data.

You can use this procedure where an exercise step states "Ensure that existing errors have been repaired".

Exercise One: Creating an Inverse and Examining Metadata

In this exercise, you will create a schema having a simple 1-M (one-to-many) inverse and observe elements of the metamodel that Jade creates to describe the user model. This schema will later be intentionally damaged and then repaired using the Logical Certifier.

1. Create a new schema called **LogCertTester**.
2. Create classes **Company** and **Product**.
3. Add public property **Product::productPrice** of type **Real**.
4. Add public key property **Product::productDescription** of type **String [30]**.
5. Create class **AllProductsByDescription** as a subclass of **MemberKeyDictionary**.

   This subclass should have a membership of type **Product** and be keyed by **Product::productDescription**.
6. Add an auto reference **Company::allMyProducts** of type **AllProductsByDescription** and a manual public inverse **Product::myCompany**.
7. Define and then save the following **JadeScript::setup** method.

```
vars
    prod : Product;
    comp : Company;
begin
    beginTransaction;
    create comp;
    create prod;
    prod.productDescription := "Box of nails";
    prod.productPrice := 12;
    prod.myCompany := comp;
    create prod;
    prod.productDescription := "2m ladder";
    prod.productPrice := 260;
```

```
        prod.myCompany := comp;
        commitTransaction;
    end;
```

8. Run the **JadeScript::setup** method and then use the inverse to navigate the objects in the Jade Object Inspector, to demonstrate that referential integrity is correct.
9. Inspect the **Company** class in the Jade Object Inspector, by running the code **Company.inspect();** inside a Workspace.
10. Select **properties** in the left pane of the inspector. Note that the type is **PropertyNDict** and that it contains **allMyProducts**.
11. Double-click on **properties** and then double-click **allMyProducts**.

    Note The oid **1000014.x** in the title bar. **1000014** is the class number of the **ExplicitInverseRef** class.

12. Select the **inverses** property. Note that it has type **InverseSet** and that it contains one inverse.
13. Double-click on **inverses**. Note that the **Inverse** object has oid **1000035.x**, where **1000035** is the class number of the **Inverse** class.
14. Extract the schema, for use in later exercises,

You will now have observed some of the glue that is the metamodel that defines the user schema.

In the next exercise, referential integrity will be intentionally broken so that we can demonstrate using the Logical Certifier to analyze and repair the damage.

Exercise Two: Breaking Referential Integrity

Jade normally ensures that referential integrity is maintained. However, as discussed earlier, the Logical Certifier repair mechanism necessarily disables inverse maintenance, which provides the ability to break referential integrity through the use of a hand-crafted, erroneous fix file.

1. Open a new Workspace and then define and run the following method to generate an erroneous fix.

```
vars
    fixLine : String;
begin
    //Remove first instance of Product from company's collection of products
    fixLine := "FIX1: remove ";
    fixLine := fixLine & AllProductsByDescription.number.String;
    fixLine := fixLine & ".1." & Company.number.String;
    fixLine := fixLine & ".2.1 ";
    fixLine := fixLine & Product.firstInstance.getOidString;
    write fixLine;
end;
```

The output from this method should have the following form. (These class numbers will vary, depending on what other schemas have been loaded.)

```
FIX1: remove 2976.1.2973.2.1 2975.1
```

In this fix:

- 2976 is the class number of the **AllProductsByDescription** class
- 2973 is the class number of the **Company** class

- 2975 is the class number of the **Product** class

2. If required, save the Workspace.
3. Create a **_logcert.fix** fix file and then paste the output from step 1 of this instruction into that file.
4. Save the file to a directory where the output files will be created; for example, **c:\temp**.
5. Close the Jade database.
6. Run the Logical Certifier application in GUI or in batch mode, directing the application to the fix file created in the previous step of this instruction.

   If you are using:

   - The GUI application, select **Repair** in the **Operation** combo box and then browse to the location of the fix file.
   - A batch file, modify the example provided in the "Running the Logical Certifier" section, earlier in this document, so that **logDirPath=c:\temp** (if that is where you saved the erroneous fix file) and **operation=repair**.

   The repair will have renamed **logCert.fix** to **logcertBackup.fix** and generated:

   - **repair.err**, which contains entries indicating repairs that failed. No errors should be listed.
   - **repair.log**, which shows repair actions that were performed. This log should end with **Repair complete: 0 errors (0 warnings)**.
   - **_logcert.fix**, which should be empty.
7. Examine the error file and log file to ensure that the repair completed successfully. The repair log should contain a line similar to the following.

   ```
   FIX1 Removed Product/2975.1 from collection
   AllProductsByDescription/2976.1.2973.2.1
   ```
8. Open the Jade Platform development environment and use the Object Inspector to verify that the **Product** instance has been removed from the company's products.

   Verify that the removed **Product** instance has a valid reference to its company. This inconsistency shows that referential integrity is broken.
9. Save the generated files (for example, to **c:\temp\firstCorruption**) and then close the database.

In the next exercise, the Logical Certifier will be used to identify the (artificially) induced anomaly.

Exercise Three: Analyzing the Damage

In this exercise, the Logical Certifier will be used to analyze the inverse relationship between **Company** and **Product**.

1. Run the Logical Certifier application in the GUI or the batch mode. Direct the application to the appropriate location where the output will be saved. If you are using:

   - The GUI application, select **Certify Selected Schemas** in the **Operation** combo box and select **LogCertTester** in the **Schema** list box.
   - A batch file, modify the example provided in the "Running the Logical Certifier" section, earlier in this document, so that **logDirPath=c:\temp** (if that is where you saved the erroneous fix file) and **operation=certify**.

2. Observe that the following four files have been produced in the specified output directory.
   - **logcert.err**, which identifies the errors that were discovered
   - **logcert.cls**, which indicates which classes have instances that require repair
   - **logcert.fix**, which is the file that will be used in repair mode to repair the identified error
   - **logcert.log**, which is the file that records the analysis procedure

Exercise Four: Interpreting the Analysis Output

When you have completed Exercise Three, an error file is produced (that is, **_logcert.err**), which contains lines in the following format.

```
---- Errors for LogCertTester::Product --------------
*** Error 1: Collection does not include reference to inverse object
Company/2973.1->Company::allMyProducts (inverse of
Product/2975.1->Product::myCompany) does not include Product/2975.1 (error
coll=AllProductsByDescription/2976.1.2973.2.1)
FIX1: add 2976.1.2973.2.1 2975.1
CHECK: '2975.1'.asOid.getPropertyValue('myCompany').Object='2973.1'.asOid and not
'2973.1'.asOid.getPropertyValue('allMyProducts').Object.Collection.includes
('2975.1'.asOid)
---- Summary
--------------------------------------------------------------------------------
    Product/2975.1->Product::myCompany [1]
```

The problem is clearly identified by the statement indicating that a product instance is not included in the collection of products held by a company instance. The fix of adding the product instance is also given. This is informational, only.

A possible runtime application consequence is that the product will be absent from a list box that displays products for the company and thus be inaccessible for updating.

Note The error is identified as being of type **Error 1**. The classification of error types is documented in "Logical Certifier Errors and Repairs", in Chapter 5 of the *Object Manager Guide*.

Open the fix file; that is, **_logcert.fix**. This is the file that is used by the Logical Certifier to perform the repair. It contains a line in the following format.

```
FIX1: add 2976.1.2973.2.1 2975.1
CHECK: '2975.1'.asOid.getPropertyValue('myCompany').Object='2973.1'.asOid and not
'2973.1'.asOid.getPropertyValue('allMyProducts').Object.Collection.includes
('2975.1'.asOid)
```

Note the presence of the **CHECK** phrase, following the repair action. This check is used by the Logical Certifier to verify that the condition identified during the analysis phase is still present in the database when the repair is run. This is important because the analysis phase would typically be performed on a backup copy of the database due to the length of time required. It also offers some protection against the possibility of running a repair when the analysis was performed against a different system.

The informational log (that is, **_logcert.log**), among other things, contains the following information.

- A description of what relationships were checked.
- A description of the memory requirements for checking each relationship. Increasing transient cache and interpreter cache will generally result in reduced analysis times.

- The number of instances of each class checked. If an exception was generated that prevented completion of analysis for a specific class, you can compare this figure with the total number of instances.
- The number of errors detected.

The **_logcert.cls** file defines the set of classes in which errors were detected. You can rename this file to **_logcert.in** and use it to perform a subsequent restricted analysis.

Exercise Five: Repairing the Damage

To run the repair, repeat step 6 in Exercise Two. Use the logs to verify that the repair completed successfully.

Use the Jade Platform development environment Object Inspector to independently verify that logical integrity has been restored. Alternatively, run another Logical Certifier analysis and verify that no errors are reported.

Exercise Six: Additional Referential Integrity Examples

Exercises Three through Five demonstrated analysis, interpretation, and repair of a specific referential integrity problem identified as Error 1.

In this exercise, other error types are investigated so that you can see how these are reported and repaired. Parts A, B, and C demonstrate the effect of inverse update mode that was discussed in "Analysis and Repair Process", earlier in this document.

For more details about the examples in this exercise, see the following subsections.

Part A: Nulling the Manual Side

1. Ensure that existing errors have been repaired.

   As mentioned earlier, you should remove the schema, restart Jade, and then reload the schema before creating fresh data.

2. This time we will intentionally break referential integrity by using the following Workspace script to generate a different erroneous fix.

   ```
   vars
       fixLine : String;
   begin
       // A product does not know its owning company
       fixLine := "FIX1: null " & Product.firstInstance.getOidString
                   & " myCompany";
       write fixLine;
   end;
   ```

   A consequence of this anomaly is that a 1090 exception *(Attempted access via null object reference)* would be raised if you attempt to reference **Product::myCompany** from this instance of **Product**.

3. Run the erroneous repair created from the script in the previous step.

4. Analyze the damage by running the Logical Certifier in analysis mode. This anomaly will be reported in the following way.

   ```
   *** Error 15: No inverses found for Company/2973.1->Company::allMyProducts
   (coll1=AllProductsByDescription/2976.1.2973.2.1, member=Product/2975.1)
   (2976.1.2973.2.1 created 01 January 1970, 12:00:00, parent 2973.1 created 17
   ```

```
July 2009, 14:48:39)
FIX1: remove 2976.1.2973.2.1 2975.1
CHECK: 2973.1'.asOid.getPropertyValue
('allMyProducts').Object.Collection.includes('2975.1'.asOid)
```

Note The repair action to perform a remove is a consequence of the **Product::myCompany** reference being defined as manual, so the assumption made by the Logical Certifier is that the reference was intentionally set to null and for some reason, inverse maintenance failed and the object was not removed.

Part B: Reverse Update Mode

1. Ensure that you have repaired existing errors.
2. Change the update mode of **Company::allMyProducts** from automatic to manual.
3. Adjust the **JadeScript::setup** method to perform a dictionary addition **Company::allMyProducts::add** as opposed to setting **Product::myCompany**, and then run setup.
4. Run the erroneous fix from "Part A", in the previous section.

The anomaly is now reported in the following way.

```
---- Errors for LogCertTester::Company --------------
*** Error 14a: Collection member does not reference inverse object
    Product/2975.1->Product::myCompany (inverse of Company::allMyProducts) does
not reference Company/2973.1 (references Object/0.0)
FIX1: set 2975.1 myCompany 2973.1  CHECK: '2973.1'.asOid.getPropertyValue
('allMyProducts').Object.Collection.includes('2975.1'.asOid) and
'2975.1'.asOid.getPropertyValue('myCompany').Object=null
```

As in Part A, the manual side of the inverse is assumed to be correct and the repair is performed against the automatic side, to restore referential integrity.

Part C: Indeterminate Update Mode Results in Choices

1. Ensure that existing errors have been corrected.
2. Change the update mode of **Company::allMyProduct** from automatic to manual/automatic.
3. Run the **JadeScript::setup** method (no change to this method is required).
4. Run the erroneous fix from Part A.

The anomaly is now reported in the following way.

```
---- Errors for LogCertTester::Company --------------
*** Error 15: No inverses found for Company/2973.1->Company::allMyProducts
(coll1=AllProductsByDescription/2976.1.2973.2.1, member=Product/2975.1)
(2976.1.2973.2.1 created 01 January 1970, 12:00:00, parent 2973.1 created 20 July
2009, 16:54:32)
//FIX1: Choose either first fix if man/auto reference Company::allMyProducts acts
as auto else following 1 line(s)
//FIX1: remove 2976.1.2973.2.1 2975.1  CHECK: '2973.1'.asOid.getPropertyValue
('allMyProducts').Object.Collection.includes('2975.1'.asOid)
//FIX1: set 2975.1 myCompany 2973.1  CHECK: '2973.1'.asOid.getPropertyValue
('allMyProducts').Object.Collection.includes('2975.1'.asOid)
```

Various fix alternatives are presented. To activate the appropriate fix, you must remove the comment (that is, *//*) in the **_logcert.fix** file.

Part D: Dictionary that Has a Member at Invalid Keys

If you run the erroneous repair from Exercise Six, Part A, it will prevent the automatic dictionary from being updated when a member's key is changed.

If you change the key property of the member, dictionary maintenance will not occur so the member will be stored at the old, incorrect key value. You can then run a repair that resets the reference **Product::myCompany**. The end state then is one in which the problem is purely that a member is stored at an incorrect key.

1. Ensure that existing errors have been corrected.
2. Run the erroneous repair created in Part A.
3. Run the following Workspace script to alter the key property values of the first instance of **Product**.

```
beginTransaction;
    Product.firstInstance.productDescription := "Box of screws";
commitTransaction;
```

4. Obtain the fix to reset **Product::myCompany**, by running the following script.

```
vars
    fixLine : String;
begin
    fixLine := "FIX1: set " & Product.firstInstance.getOidString
                & " myCompany " & Company.firstInstance.getOidString;
    write fixLine;
end;
```

   The form of the fix should be as follows.

```
FIX1: set 2975.1 myCompany 2973.1
```

5. Run the repair constructed from the output of the previous step.

   Note that attempting to retrieve the first product, a box of screws, via the key value **"Box of screws"** results in a null object referenced being returned.

```
Company.firstInstance.allMyProducts.getAtKey("Box of screws")
```

   Conversely, attempting to retrieve a "**Box of nails"** results in a "**Box of screws"** being returned.

6. Run the Logical Certifier in analysis mode. This anomaly will be reported in the following manner.

```
---- Errors for LogCertTester::Company --------------
*** Error 3: Collection contains object at invalid keys
    AllProductsByDescription/2976.1.2973.2.1 Company::allMyProducts contains
Product/2975.1 at invalid keys (2976.1.2973.2.1 created 01 January 1970,
12:00:00, parent 2973.1 created 05 August 2009, 16:37:59)
FIX1: rebuild 2976.1.2973.2.1
```

After running the repair, the first product will be accessible via its **"Box of screws"** key value.

Note The Logical Certifier will also report a second error, stating that the **Product** instance cannot be found in the collection. The nature of the corresponding fix depends on the update mode specified for the inverse. This second fix will not be run, however, since its execution is conditional upon still being unable to find the product in the collection, and the rebuild fix will be run first to address this. If an error prevents the first fix from being run, the second fix will then be run and the update mode determines the final outcome. This is the subject of the next exercise.

Part E: Repair that Fails due to Duplicate Entry

If you run the erroneous repair from Exercise Six, Part A, it will prevent the automatic dictionary from being updated when a member's key is changed. If you change the key property of the member to be the same as that of another member, dictionary maintenance will not occur. You can then run a repair that resets the reference **Product::myCompany**.

The end state is then one in which the member is stored at an incorrect key. However, this is different from Part D, as there is the additional fact that the key property value is the same as that of another member so attempting to insert the member at its key property value will cause a duplicate key exception and the rebuild action will be aborted. In contrast to Part D, the second fix generated will run as a consequence of the first fix not being implemented.

1. Ensure that existing errors have been corrected.
2. Run the erroneous repair created in Part A.
3. Run the following Workspace script to alter the key property values of the first instance of **Product**.

```
beginTransaction;
    Product.firstInstance.productDescription :=
        Product.lastInstance.productDescription;
commitTransaction;
```

4. Obtain the fix to reset the **Product::myCompany** by running the following script.

```
vars
    fixLine : String;
begin
    fixLine := "FIX1: set " & Product.firstInstance.getOidString
               & " myCompany " & Company.firstInstance.getOidString;
    write fixLine;
end;
```

   The form of the fix should be as follows.

```
FIX1: set 2975.1 myCompany 2973.1
```

5. Run the repair constructed from the output of the previous step.
6. Verify in the Object Inspector that both **Product** instances now have the same name and that the reference to the parent company is set.
7. Run the Logical Certifier in analysis mode.

   Save the output in a new folder named according to the update mode for the inverse.
8. Run the repair.

Observe that the rebuild fix to address the invalid key fails due to a duplicate key being encountered. This is documented in the repair error log, as follows.

```
Error: FIX1 rebuild AllProductsByDescription/2976.1.2973.2.1: size before=2,
size after=2 results in duplicate
```

The end state of the system will be determined by the second fix, which in turn depends on the update mode in place. At this point, you would typically evaluate the situation and determine that the two errors reported in the analysis phase come from the same condition (an object at the wrong key). You may be able to determine that the key property value is incorrect (as is the case here) and the correct value from other information (the product price, in this example).

You would then set the correct key property value (for example, by a JadeScript) and then go ahead with the rebuild, if required.

9. Verify that the second error reported (and corresponding fix) will vary, according to the update mode (**Company <-> Product**), as follows.

   - Collection is automatically updated. (Both fixes will fail.)

     ```
     ---- Errors for LogCertTester::Product --------------
     *** Error 1: Collection does not include reference to inverse object
         Company/2973.1->Company::allMyProducts (inverse of
     Product/2975.1->Product::myCompany) does not include Product/2975.1
     (error coll=AllProductsByDescription/2976.1.2973.2.1)
     FIX2: add 2976.1.2973.2.1 2975.1  CHECK: '2975.1'.asOid.getPropertyValue
     ('myCompany').Object='2973.1'.asOid and not
     '2973.1'.asOid.getPropertyValue
     ('allMyProducts').Object.Collection.includes('2975.1'.asOid)
     ```

   - Reference **Product::myCompany** is automatically updated. (Removes part of the relationship.)

     ```
     ---- Errors for LogCertTester::Product --------------
     *** Error 8: No inverses found for Product/2975.1->Product::myCompany
     (obj=Product/2975.1, related=Company/2973.1) (2975.1 created 22 September
     2009, 11:32:51)
     FIX2: null 2975.1 myCompany  CHECK: '2975.1'.asOid.getPropertyValue
     ('myCompany').Object='2973.1'.asOid
     ```

   - Both sides man/auto. (No outcome unless a choice is made. Choices match first two cases.)

     ```
     ---- Errors for LogCertTester::Product --------------
     *** Error 8: No inverses found for Product/2975.1->Product::myCompany
     (obj=Product/2975.1, related=Company/2973.1) (2975.1 created 22 September
     2009, 11:32:51)
     //FIX2: Choose either first fix if man/auto Product::myCompany acts as
     auto else following 1 line(s)
     //FIX2: null 2975.1 myCompany  CHECK: '2975.1'.asOid.getPropertyValue
     ('myCompany').Object='2973.1'.asOid
     //FIX2: add 2976.1.2973.2.1 2975.1  CHECK:
     '2975.1'.asOid.getPropertyValue('myCompany').Object='2973.1'.asOid and
     not '2973.1'.asOid.getPropertyValue
     ('allMyProducts').Object.Collection.includes('2975.1'.asOid)
     ```