

Understanding Sun Master Index Processing

Beta

Part No: 821-1057-10
March 2010

Copyright ©2010 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Understanding Sun Master Index Processing	7
About Sun Master Index	7
 Understanding Operational Processes	 9
Learning About Message Processing	9
Inbound Message Processing	10
About Inbound Messages	11
Outbound Message Processing	12
About Outbound Messages	13
Inbound Message Processing Logic	15
Primary Function Processing Logic	23
activateEnterpriseObject	23
activateSystemObject	23
addSystemObject	24
createEnterpriseObject	25
deactivateEnterpriseObject	25
deactivateSystemObject	25
deleteSystemObject	26
mergeEnterpriseObject	27
mergeSystemObject	28
transferSystemObject	31
undoAssumedMatch	33
unmergeEnterpriseObject	34
unmergeSystemObject	36
updateEnterpriseDupRecalc	38
updateEnterpriseObject	39
updateSystemObject	40

The Database Structure	43
About the Database	43
Database Table Details	45
SBYN_OBJECT_NAME	46
SBYN_OBJECT_NAMESBR	46
SBYN_CHILD_OBJECT	47
SBYN_CHILD_OBJECTSBR	47
SBYN_APPL	48
SBYN_ASSUMEDMATCH	48
SBYN_AUDIT	49
SBYN_COMMON_DETAIL	50
SBYN_COMMON_HEADER	51
SBYN_ENTERPRISE	51
SBYN_MERGE	52
SBYN_OVERWRITE	52
SBYN_POTENTIALDUPLICATES	53
SBYN_SEQ_TABLE	54
SBYN_SYSTEMOBJECT	56
SBYN_SYSTEMS	57
SBYN_SYSTEMSBR	58
SBYN_TRANSACTION	59
SBYN_USER_CODE	60
Sample Database Model	61
Working with the Java API	65
Overview	65
Java Class Types	65
Static Classes	66
Dynamic Web Services Classes	66
Dynamic Web Service Methods	66
Dynamic API Method Descriptions	67
Web Services Helper Classes	85
Enterprise Object Classes	85
SystemObjectName	95
Parent Beans	100

Child Beans	108
DestinationEO	113
MergeObjectNameResult	114
SBRObjecName	117
SearchObjectNameResult	121
SourceEO	124
SystemObjecNamePK	126
 Inbound Message Processing with Custom Logic	 129
Custom Decision Point Logic	129
 A Match Types and Field Names	 133
Match Types and Standardization Types	133
Sun Match Engine Match Types	134
Person Match Types	134
BusinessName Match Types	135
Address Match Types	136
Miscellaneous Match Types	137

Understanding Sun Master Index Processing

The topics listed here provide conceptual information about standard processing logic for the master index application, how data flows through the master index application, the database structure, and the dynamic Java API.

- [Understanding Operational Processes](#)
- [The Database Structure](#)
- [Working with the Java API](#)
- [Inbound Message Processing with Custom Logic](#)
- [Match Types and Field Names](#)

About Sun Master Index

Sun Master Index provides a flexible framework that allows you to create matching and indexing applications called *enterprise-wide master index applications*. It is an application building tool to help you design, configure, and create a master index application that will uniquely identify and cross-reference the business objects stored in your system databases. Business objects can be any type of entity for which you store information, such as customers, patients, vendors, businesses, inventory, and so on.

When you create a Sun master index application, custom database scripts and a custom Java API are automatically generated based on the information you specify in the wizard and the configuration files. Both the database scripts and API are derived from the object structure you define. For example, if you create a master index application with an Address object, the database scripts will define a table named SBYN_ADDRESS and one named SBYN_ADDRESSESSBR. The Java API will include a class named AddressObject that includes “get” methods for each field you defined for the Address object.

Sun Master Index provides features and functions to allow you to create and configure master index application for any type of data. The primary function of Sun Master Index is to automate the creation of a highly configurable master index application. A wizard guides you through the initial setup steps, and the Configuration Editor allows you to further customize the

configuration of the master index application. The components you need to implement a master index application are automatically generated.

Sun Master Index provides the following features:

- **Rapid Development** - Rapid and intuitive development of a master index application using a wizard to create the master index configuration and using XML documents to configure the attributes of the index. Templates are provided for quick development of person and company object structures.
- **Automated Component Generation** - Sun Master Index automatically creates the configuration files that define the primary attributes of the master index application, including the configuration of the Master Index Data Manager (MIDM). Sun Master Index also generates scripts that create the appropriate database schemas and an XSD based on the object structure you create and configure.
- **Configurable Survivor Calculator** - Sun Master Index provides predefined strategies for determining which field values to populate in the single best record (SBR). You can define different survivor rules for each field, and you can create a custom survivor strategy to implement in the master index application.
- **Flexible Architecture** - Sun Master Index provides a flexible platform that allows you to create a master index application for any business object. You can customize the object structure so the master index application can match and store any type of data, allowing you to design an application that specifically meets your data processing needs.
- **Configurable Matching Algorithm** - Sun Master Index provides standard support for the Sun Match Engine. In addition, you can plug in a custom matching algorithm to the master index application.
- **Custom Java API** - Sun Master Index generates a Java API that is customized to the object structure you define.
- **Standard Reports** - Sun Master Index provides a set of standard reports with each master index application that can be run from a command line or from the MIDM. The reports help you monitor the state of the data stored in the master index application and help you identify configuration changes that might be required.

Understanding Operational Processes

Master index applications created by Sun Master Index use a custom Java API library to transform and route data into and out of the master index database. In order to customize the way the Java methods transform the data, it is helpful to understand the logic of the primary processing functions and how messages are typically processed through the master index system.

This chapter describes and illustrates the processing flow of messages to and from the master index application, providing background information to help design and create custom processing rules for your implementation. This chapter discusses the following topics:

- [“Learning About Message Processing” on page 9](#)
- [“Inbound Message Processing” on page 10](#)
- [“Outbound Message Processing” on page 12](#)
- [“Inbound Message Processing Logic” on page 15](#)
- [“Primary Function Processing Logic” on page 23](#)

Learning About Message Processing

This section provides a summary of how inbound and outbound messages can be processed in a master index application. A master index application cross-references records stored in various computer systems of an organization and identifies records that might represent or do represent the same object. The master index application uses web services to connect to and share data with these external systems.

[Figure 1](#) illustrates the flow of information through a master index application that includes a JMS Topic to which updates to the index are published.

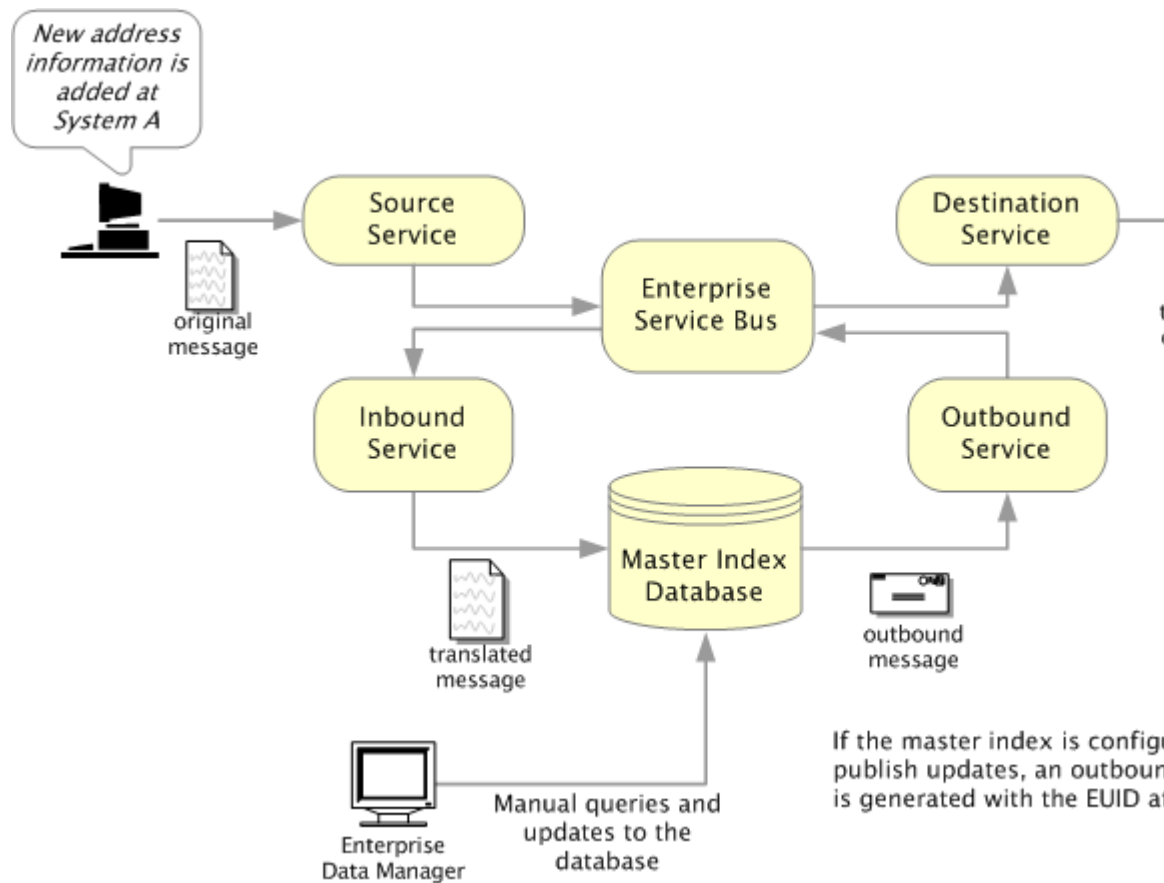


FIGURE 1 Master Index Processing Flow

Inbound Message Processing

An inbound message refers to the transmission of data from external systems to the master index database. These messages can be sent into the database via a number of web services. Inbound messages can be stored in journal files and tracked in the log files. The steps below describe how inbound messages are processed.

1. Messages are created in an external system, and the message is transmitted to the Enterprise Service Bus (ESB) via that system's binding service.
2. The ESB identifies the message and the appropriate process to which the message should be sent. The message is then routed for processing.

3. The message is modified into the appropriate format for the master index database, and validations are performed against the data elements of the message to ensure accurate delivery. The message is typically validated using the business process logic and other information stored in the master index configuration files.
4. If the message was successfully transmitted to the database, the appropriate changes to the database are processed.
5. After the master index application processes the message, an enterprise-wide universal identifier (EUID) is returned (for a new or updated record). That EUID can be sent back out through a different web service to the external system. Alternatively, the entire updated message can be published using the generated outbound message (see [“Outbound Message Processing” on page 12](#)).

Figure 2 below illustrates the flow of a message inbound to an Sun Master Index application.

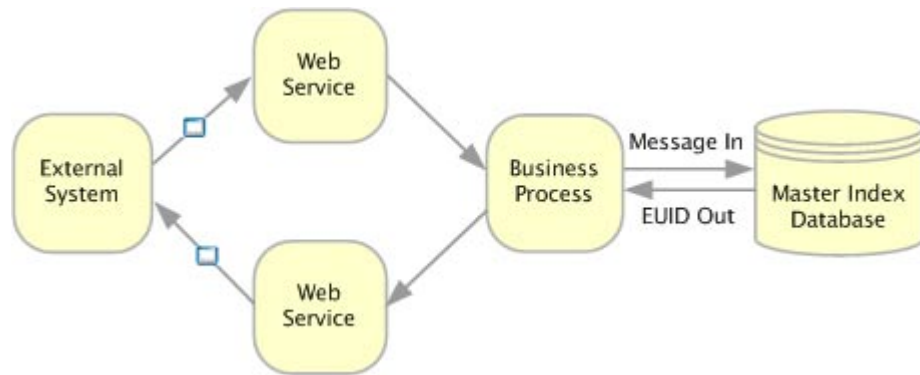


FIGURE 2 Inbound Message Processing Data Flow

About Inbound Messages

The format for the bulk of the inbound message is defined by the object structure of the master index, located in the `object.xml` configuration file. The inbound messages can either conform to the required format for the master index application, or they can be mapped to the correct format in a business process.

In addition to the objects and fields defined in `object.xml`, you can include standard master index application fields. For example, you must include the system and local ID fields, and you can also include transaction information, such as the date and time of the transaction, the transaction type, user ID, and so on. Transaction fields include the following:

- Message Id
- EventTypeCode

- UserId
- AssigningSystem
- Source
- Department
- TerminalId
- DateOfEvent
- TimeOfEvent

If you do not send these fields into the master index application, default values are used (for example, the date and time fields default to the date and time the transaction is processed by the master index application).

Outbound Message Processing

An outbound message refers to the transmission of data from the master index database to any external system. Messages can be transmitted from the master index application in two ways. The first way is by transmitting the output of `executeMatch` (an EUID). This is described in [“Inbound Message Processing” on page 10](#) and is only used for messages received from external systems.

The second way is by publishing updates from the master index application to a JMS Topic, which allows you to publish complete, updated single best records (SBRs) to any system subscribing to that topic. When updates are made to the database from either external systems or the Master Index Data Manager (MIDM), the master index application generates outbound messages in the format of the outbound XSD.

Note – A Sun master index application only publishes the outbound message to JMS Topics and not to JMS Queues.

This section describes how the second type of outbound message is processed. A JMS Topic must be defined in the application server for this type of processing to occur.

1. When a message is received from an external system or data is entered through the MIDM, the master index application processes the information and generates an XML message, which is sent to the JMS Topic that is configured to publish messages from the master index application.
2. Messages published by the JMS Topic are processed through a business process (or other client application), which uses the master index outbound XSD. The business process transforms the message into the appropriate format.
3. The message is routed using the appropriate binding component and sent to the appropriate external systems.

Figure 3 below illustrates the flow of data for a message outbound from a master index application.

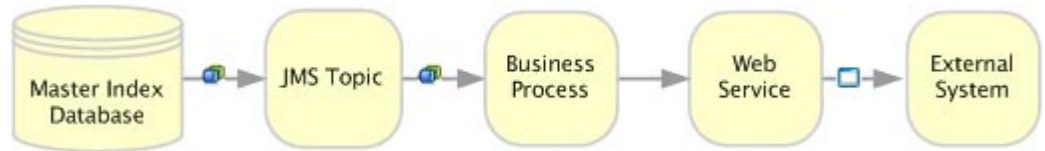


FIGURE 3 Outbound Message Processing Data Flow

About Outbound Messages

When you customize the master index object structure and generate the master index application, an outbound schema file named `outbound.xsd` is created based on the object structure. This XSD is used to publish changes in the master index database to external systems via a JMS Topic. The output of the `executeMatch` process described earlier is an EUID of the new or updated record. You can use this EUID to obtain additional information and configure a business process to output the data, or you can process all updates in the master index application through a JMS Topic using the outbound XSD.

Outbound XSD Structure

The outbound XSD is located in the file structure for the master index application under the `files-generated` folder. The XSD includes transaction information along with the updated record from the master index database, and includes the following primary elements: `OutMsg`, `SBR`, `SystemObject`, the parent object, and any child objects. The `OutMsg` element defines the Event and ID. The Event field is populated with the type of transaction that created the outbound message, and the ID field is populated with the unique identification code of that transaction. The `SBR` element is created from `object.xml`. Table 1 describes the components of the SBR portion of the outbound OTD.

TABLE 1 Outbound XSD SBR Attributes

Node	Description
EUID	The EUID of the record that was inserted or modified.
Status	The status of the record.
CreateFunction	The date the record was first created.
CreateUser	The logon ID of the user who created the record.
UpdateSystem	The processing code of the external system from which the updates to an existing record originated.

TABLE 1 Outbound XSD SBR Attributes *(Continued)*

Node	Description
ChildType	The name of the parent object.
CreateSystem	The processing code of the external system from which the record originated.
UpdateDateTime	The date and time the record was last updated.
CreateDateTime	The date and time the record was created.
UpdateFunction	The type of function that caused the record to be modified.
RevisionNumber	The revision number of the record.
UpdateUser	The logon ID of the user who last updated the record.
SystemObject	The object's local identifier in a specified system. This field has three sub-fields: LID: The local ID assigned to the person in the system of origin. System: The processing code of the system of origin. Status: The status of the local ID in the enterprise record.
Object_Name	The fields in this node are defined by the object structure (as defined in <code>object.xml</code>). It is named by the parent object and contains all fields and child objects defined in the structure. This section varies depending on the customizations made to the object structure.

Outbound Message Trigger Events

When outbound messaging is enabled, the following transactions automatically generate an outbound message that is sent to the JMS Topic (if a JMS Topic has been incorporated into the master index Project).

- Activating a system record
- Activating an enterprise record
- Adding a system record
- Creating an enterprise record
- Deactivating a system record
- Deactivating an enterprise record
- Merging an enterprise record
- Merging a system record
- Transferring a system record
- Unmerging an enterprise record
- Unmerging a system record
- Updating an enterprise record
- Updating a system record

Sample Outbound Message

The following text is a sample outbound message for a master index application based on a master person index. Your outbound messages will appear differently depending on how you configure the client Project connectivity components.

```
<?xml version="1.0" encoding="UTF-8"?>
<OutMsg Event="UPD" ID="00000000000000004010">
  <SBR EUID="0000006501" Status="active" CreateFunction="Add" ChildType="Customer"
    CreateSystem="System" UpdateFunction="Update" RevisionNumber="3" CreateUser="mdm"
    UpdateSystem="System" UpdateDate="" CreateDate="" UpdateUser="mdm">
    <SystemObject SystemCode="ORACLE" LID="2372385720" Status="active"></SystemObject>
    <SystemObject SystemCode="ORACLE" LID="2837482562" Status="active"></SystemObject>
    <SystemObject SystemCode="SAP" LID="8327423434" Status="active"></SystemObject>
    <Customer PersonCatCode="MEMBER" FirstName="ELIZABETH" FirstName_Std="ELIZABETH"
      FirstName_Phon="E421" MiddleName="ANN" LastName="WARNER" LastName_Std="WARNER"
      LastName_Phon="WARNAR" Suffix="" Title="DR" SSN="555999222"
      DOB="04/14/1964 00:00:00" Death="" Gender="F" MStatus="M" Race="C" Ethnic="31"
      Religion="AG" Language="EN" SpouseName="CRAIG" MotherName="JEN" MotherMN="SMITH"
      FatherName="MARK" Maiden="MILLER" PobCity="MILFORD" PobState="ONTARIO"
      PobCountry="CAN" VIPFlag="NONE" VetStatus="NONE" Status=""
      DriverLicense="CT12941284" DriverLicenseSt="CT" Dod="" DeathCertificate=""
      Nationality="CAN" Citizenship="CAN">
      <Alias FirstName="LIZ" MiddleName="" LastName="MILLER"></Alias>
      <Address AddressType="H" AddressLine1="1330 BLOSSOM ST."
        AddressLine1_HouseNo="1330" AddressLine1_StDir="" AddressLine1_StName="BLOSSOM"
        AddressLine1_StPhon="BLASAN" AddressLine1_StType="St" AddressLine2=""
        City="SHEFFIELD" StateCode="CT" PostalCode="09876" PostalCodeExt=""
        County="CAPE BURR" CountryCode="USA"></Address>
      <Phone PhoneType="CC" Phone="9894774477" PhoneExt=""></Phone>
    </Customer>
  </SBR>
</OutMsg>
```

Inbound Message Processing Logic

When records are transmitted to the master index application, one of the “execute match” methods is usually called and a series of processes are performed to ensure that accurate and current data is maintained in the database. The execute match methods include `executeMatch`, `executeMatchUpdate`, `executeMatchDupRecalc`, and `executeMatchUpdateDupRecalc`. The MIDM uses `executeMatchGui`. For more information about how these methods differ, refer to the Javadocs provided with Sun Master Index.

The steps performed by the standard `executeMatch` method are outlined below, and the diagrams on the following pages illustrate the message processing flow. The processing steps performed in your environment might vary from this depending on how you customize the business process and environment.

The steps outlined below refer to the following parameters in the `master.xml` file. They are described in [Sun Master Index Configuration Reference](#)).

- OneExactMatch parameter
- SameSystemMatch parameter
- MatchThreshold parameter
- DuplicateThreshold parameter
- Update mode

Note – There are several decision points in the match process that can be defined by custom logic using custom plug-ins (for more information, see [Sun Master Index Configuration Reference](#)). These decision points are not listed in the below steps, which describe the default processing logic. [Inbound Message Processing with Custom Logic](#) provides the same steps as below with the decision points included.

1. When a message is received by the master index application, a search is performed for any existing records with the same local ID and system as those contained in the message. This search only includes records with a status of **A**, meaning only active records are included. If a matching record is found, an existing EUID is returned.
2. If an existing record is found with the same system and local ID as the incoming message, it is assumed that the two records represent the same object. Using the EUID of the existing record, the master index application performs an update of the record's information in the database.
 - If the update does not make any changes to the object's information, no further processing is required and the existing EUID is returned.
 - If there are changes to the object's information, the updated record is inserted into database, and the changes are recorded in the `sbyn_transaction` table.
 - If there are changes to key fields (that is, fields used for matching or for the blocking query) and the update mode is set to pessimistic, potential duplicates are reevaluated for the updated record.
3. If no records are found that match the record's system and local identifier, a second search is performed using the blocking query. A search is performed on each of the defined query blocks to retrieve a candidate pool of potential matches.

Each record returned from the search is weighted using the fields defined for matching in the inbound message.
4. After the search is performed, the number of resulting records is calculated.
 - If a record or records are returned from the search with a matching probability weight above the match threshold, the master index application performs exact match processing (see Step 5).

- If no matching records are found, the inbound message is treated as a new record. A new EUID is generated and a new record is inserted into the database.
- 5. If records were found within the high match probability range, exact match processing is performed as follows:
 - If only one record is returned from this search with a matching probability that is equal to or greater than the match threshold, additional checking is performed to verify whether the records originated from the same system (see Step 6).
 - If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is set to false, then the record with the highest matching probability is checked against the incoming message to see if they originated from the same system (see Step 6).
 - If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is true, a new EUID is generated and a new record is inserted into the database.
 - If no record is returned from the database search, or if none of the matching records have a weight in the exact match range, a new EUID is generated and a new record is inserted into the database.

Note – Exact matching is determined by the OneExactMatch parameter, and the match threshold is defined by the MatchThreshold parameter. For more information about these parameters, see [Sun Master Index Configuration Reference](#).

- 6. When records are checked for same system entries, the master index application tries to retrieve an existing local ID using the system of the new record and the EUID of the record that has the highest match weight.
 - If a local ID is found and same system matching is set to true, a new record is inserted, and the two records are considered to be potential duplicates. These records are marked as same system potential duplicates.
 - If a local ID is found and same system matching is set to false, it is assumed that the two records represent the same object. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 earlier.
 - If no local ID is found, it is assumed that the two records represent the same object and an assumed match occurs. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 earlier.
- 7. If a new record is inserted, all records that were returned from the blocking query are weighed against the new record using the matching algorithm. If a record is updated and the update mode is pessimistic, the same occurs for the updated record. If the matching probability weight of a record is greater than or equal to the potential duplicate

threshold, the record is flagged as a potential duplicate (for more information about thresholds and the update mode, see [Sun Master Index Configuration Reference](#)).

The following flow charts provide a visual representation of the processes performed in the default configuration. [Figure 4](#) and [Figure 5](#) represent the primary flow of information. [Figure 6](#) expands on update procedures illustrated in [Figure 4](#) and [Figure 5](#).

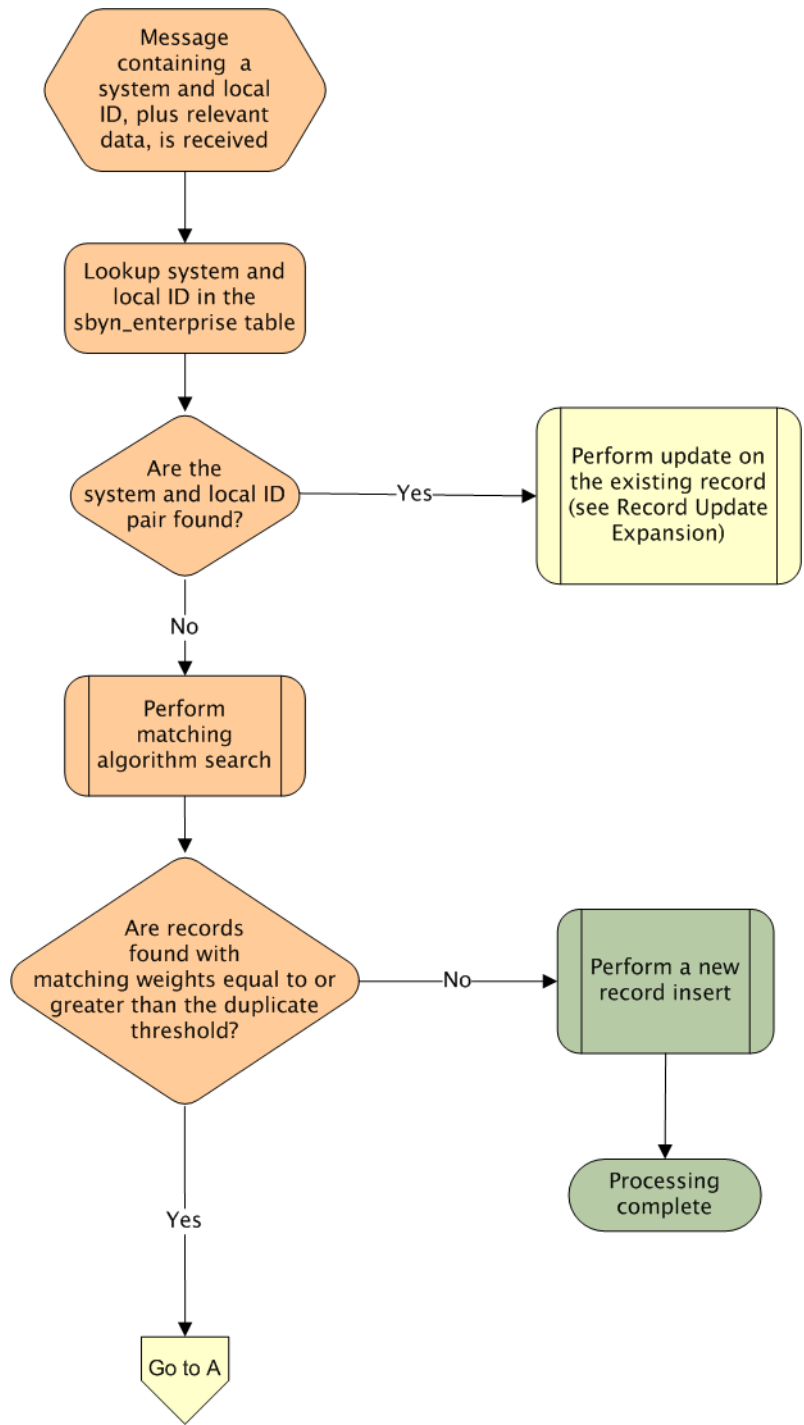
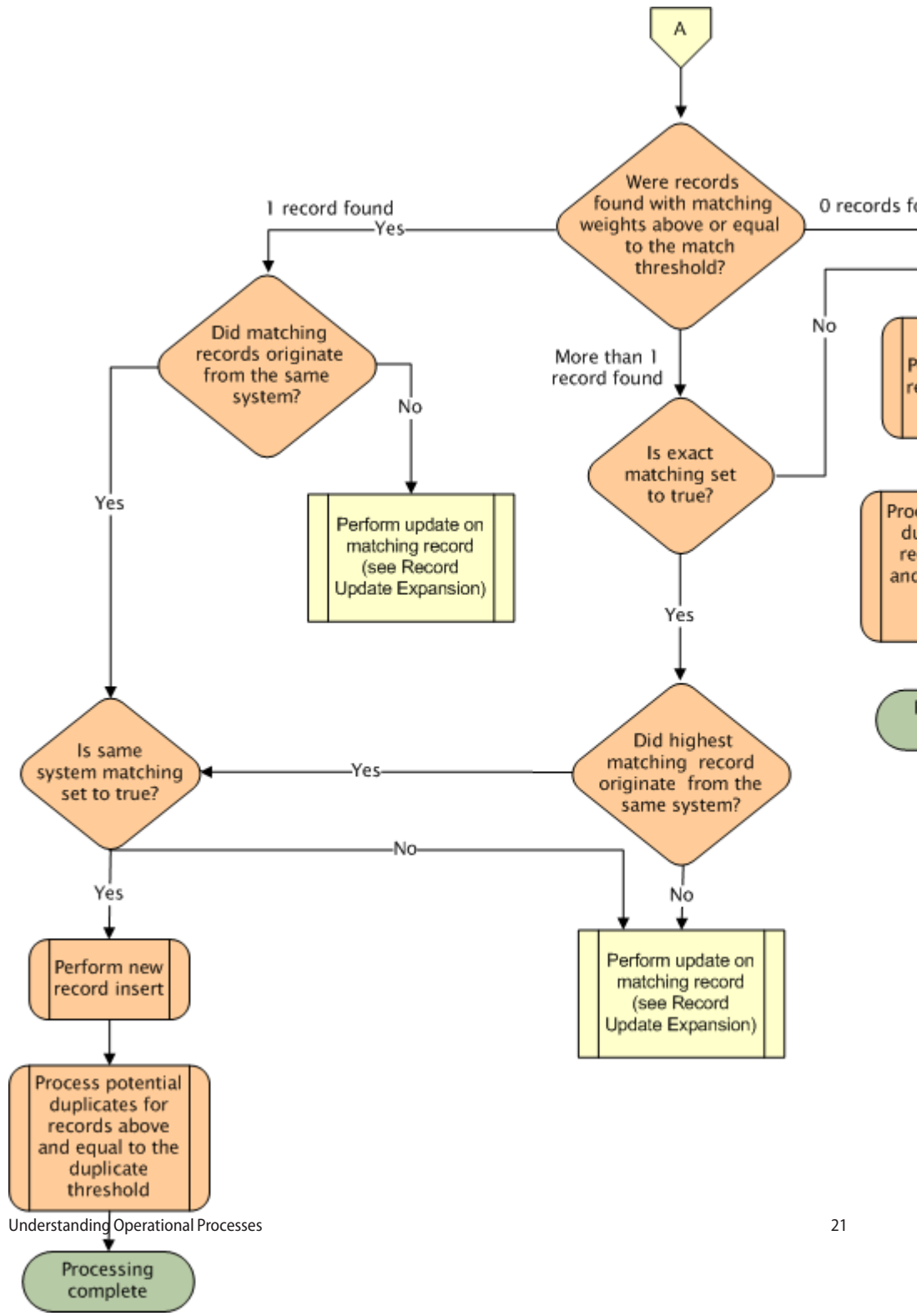
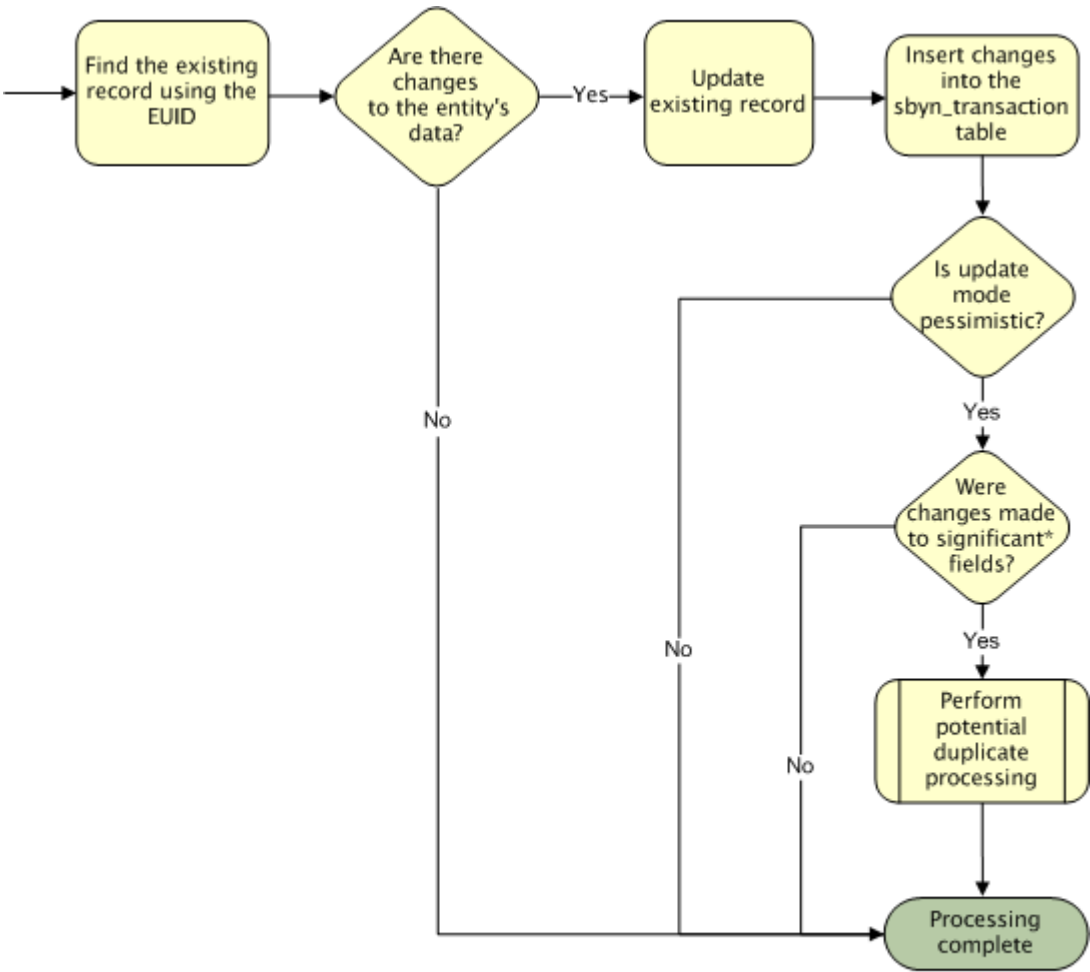


FIGURE 1 Inbound Message Processing using executeMatch





* Significant fields for potential duplicate processing include those defined for matching and those included in the blocking query used for matching

FIGURE 6 Record Update Expansion

Primary Function Processing Logic

The primary functions of a master index application can be performed from the MIDM or can be called from business processes, Java clients, web services, and so on. Whether potential duplicates are evaluated after a call to any of these functions is dependent on the update mode settings. Potential duplicates are only processed against the single best record (SBR) and not the system records. These functions are all defined in the `MasterController` and `MasterControllerEJB` classes, and are fully described in the Sun Master Index Javadocs. In the following diagrams, significant fields for potential duplicate processing include fields defined for matching and fields included in the blocking query used for matching. In all of the methods described below, an entry is made in the transaction history table (sbyn_transaction).

activateEnterpriseObject

This method reactivates an enterprise record. The MIDM calls this method when you reactivate a previously deactivated EUID. Since all potential duplicates were deleted when the EUID was originally deactivated, potential duplicates are always recalculated, regardless of the update mode. [Figure 7](#) illustrates the processing steps.

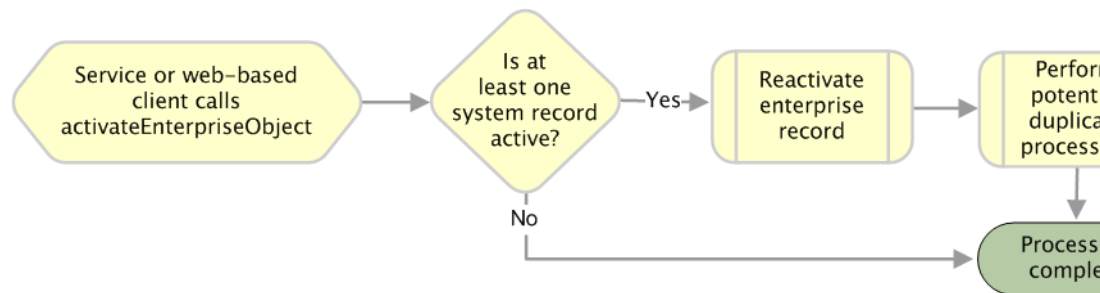


FIGURE 7 activateEnterpriseObject Processing

activateSystemObject

This method reactivates a system record. The MIDM calls this method when you reactivate a previously deactivated system record. If the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. [Figure 8](#) illustrates the processing steps.

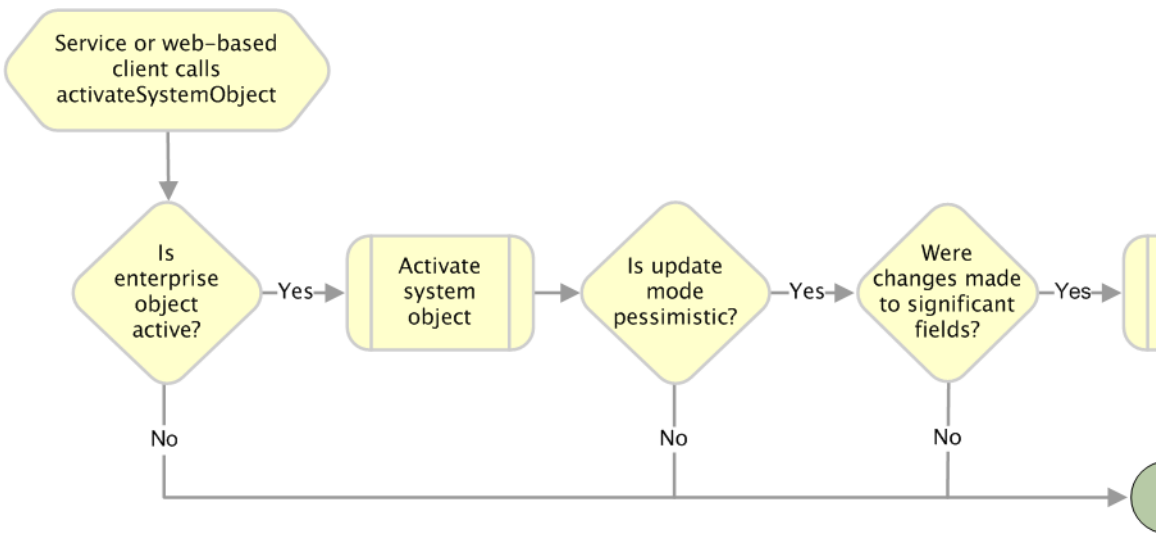


FIGURE 8 activateSystemObject Processing

addSystemObject

This method adds a system record to an enterprise record. The MIDM calls this method when you add a system record to an existing enterprise record from the Record Details window. If the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR. If key fields were updated and the update mode is set to pessimistic, potential duplicates are recalculated for the enterprise record. [Figure 9](#) illustrates the processing steps.

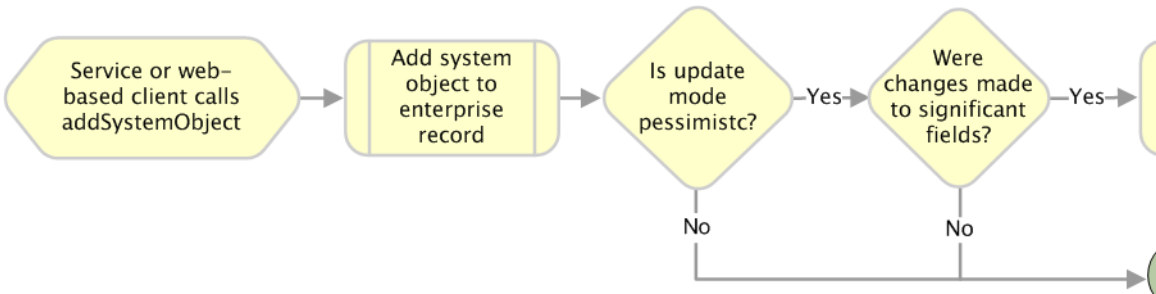


FIGURE 9 addSystemObject Processing

createEnterpriseObject

There are two `createEnterpriseObject` methods, both of which add a new enterprise record to the database and bypass any potential duplicate processing. One method takes only one system record as a parameter and the other takes an array of system records. These methods cannot be called from the MIDM and are designed for use in business processes, web services, or Java clients.

deactivateEnterpriseObject

This method deactivates an enterprise record specified by its EUID. The MIDM calls this method when you deactivate an EUID from the Record Details page. When an enterprise record is deactivated, all potential duplicate listings for that record are deleted.

deactivateSystemObject

This method deactivates a system record in an enterprise record. The MIDM calls this method when you deactivate a system record. If the enterprise record containing this system record has no active system records remaining, the enterprise record is deactivated and all potential duplicate listings are deleted. (Note that if the system record is reactivated, then the enterprise record is recreated.) If the enterprise record has active system records after the transaction and the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. [Figure 10](#) illustrates the processing steps.

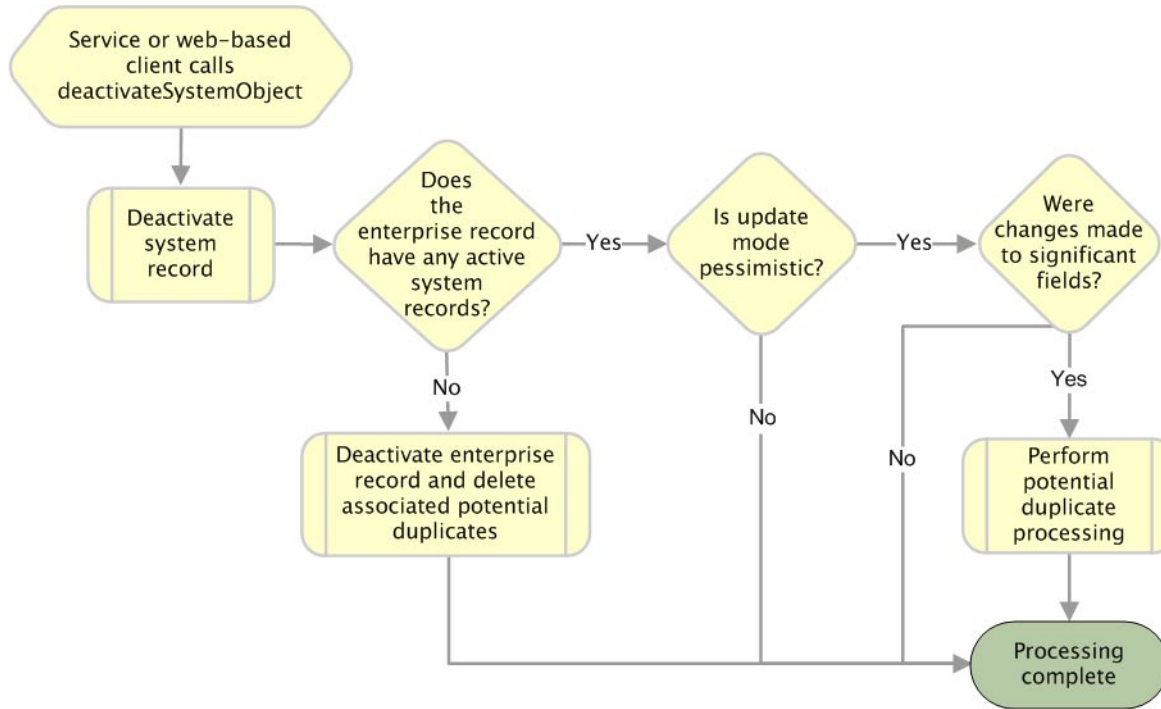
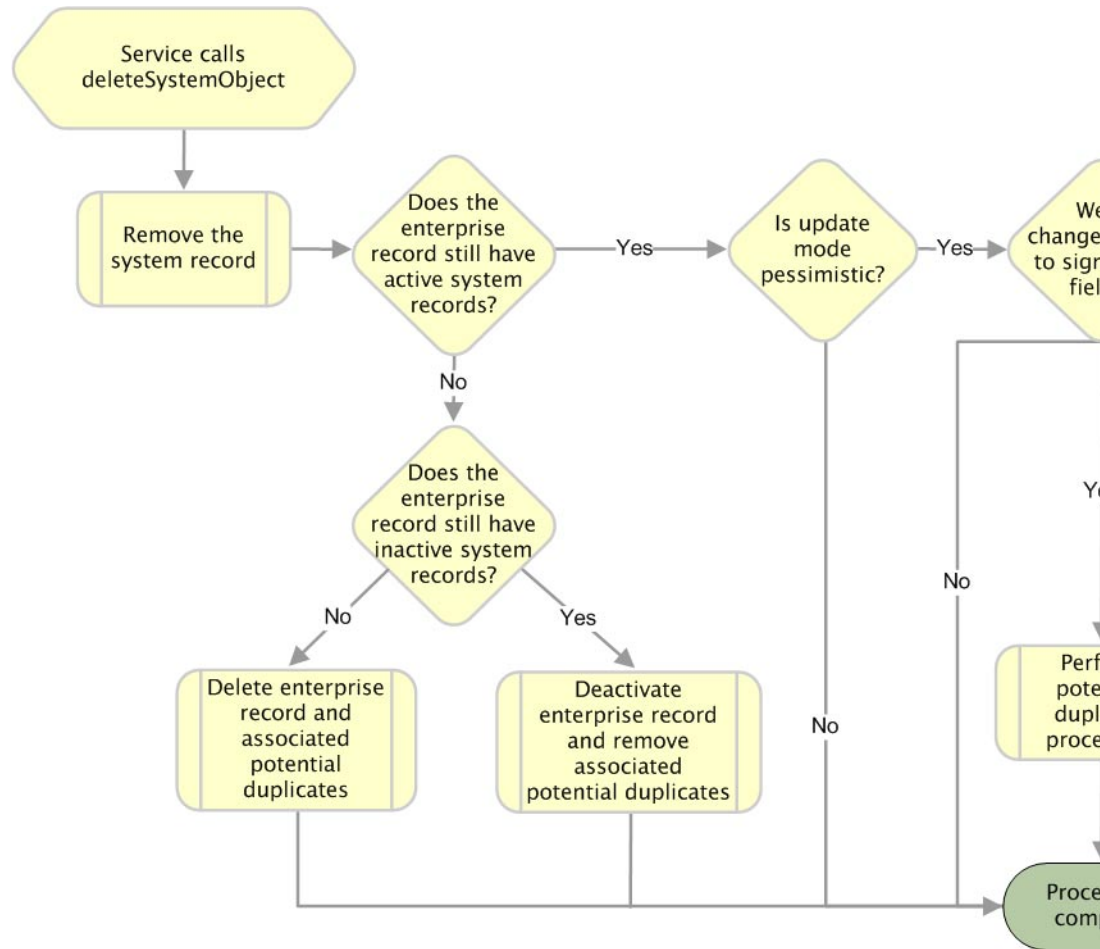


FIGURE 10 deactivateSystemObject Processing

deleteSystemObject

Unlike `deactivateSystemObject`, this method permanently removes a system record from an enterprise record. This method cannot be called from the MIDM. If the enterprise record containing the deleted system record has no active system records remaining, the enterprise record is deactivated (even if the enterprise record does have deactivated system records). If the enterprise record has no remaining system records after the system object is deleted, the enterprise record is also deleted. In both cases, any potential duplicate listings for that enterprise record are removed. If the enterprise record has active system records after the transaction and the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. [Figure 11](#) illustrates the processing steps.

FIGURE 11 `deleteSystemObject` Processing

`mergeEnterpriseObject`

There are four `mergeEnterpriseObject` methods that merge two enterprise records (see the Javadocs provided with Sun Master Index for more information about each). The MIDM calls a merge method twice during a merge transaction. When you select records to merge and then click Preview, the method is called with the `calculateOnly` parameter set to true in order to display the merge result record for you to view. When you confirm the merge, the MIDM calls this method with the `calculateOnly` parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed. The method called by the MIDM checks the SBRs of the records involved in the merge against their corresponding SBRs in the database. If the SBRs differ, the merge is not performed since that means the records were changed by someone else during the merge process.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged enterprise record to merged and deletes all potential duplicate listings for the merged enterprise record. If the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the surviving enterprise record. If key fields were updated, potential duplicates are recalculated for the enterprise record. [Figure 12](#) illustrates the processing steps, and includes the check for SBR differences, which only occurs in two of the merge methods.

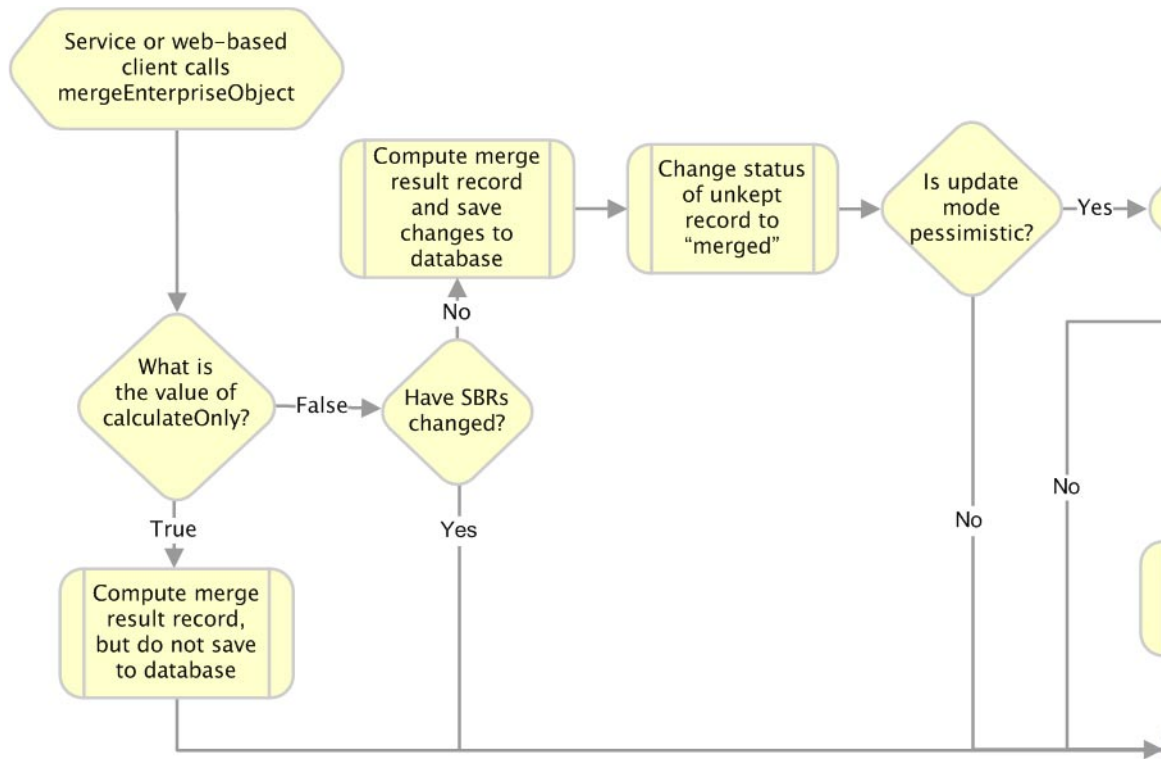


FIGURE 12 `mergeEnterpriseObject` Processing

`mergeSystemObject`

There are four methods that merge two system records that are either from the same enterprise record or from two different enterprise records (for more information about each method, see the Javadocs provided with Sun Master Index). The system records must originate from the same external system. The MIDM calls this method twice during a system record merge transaction. When you first click Keep LID# (where # is the heading number of the LID), the

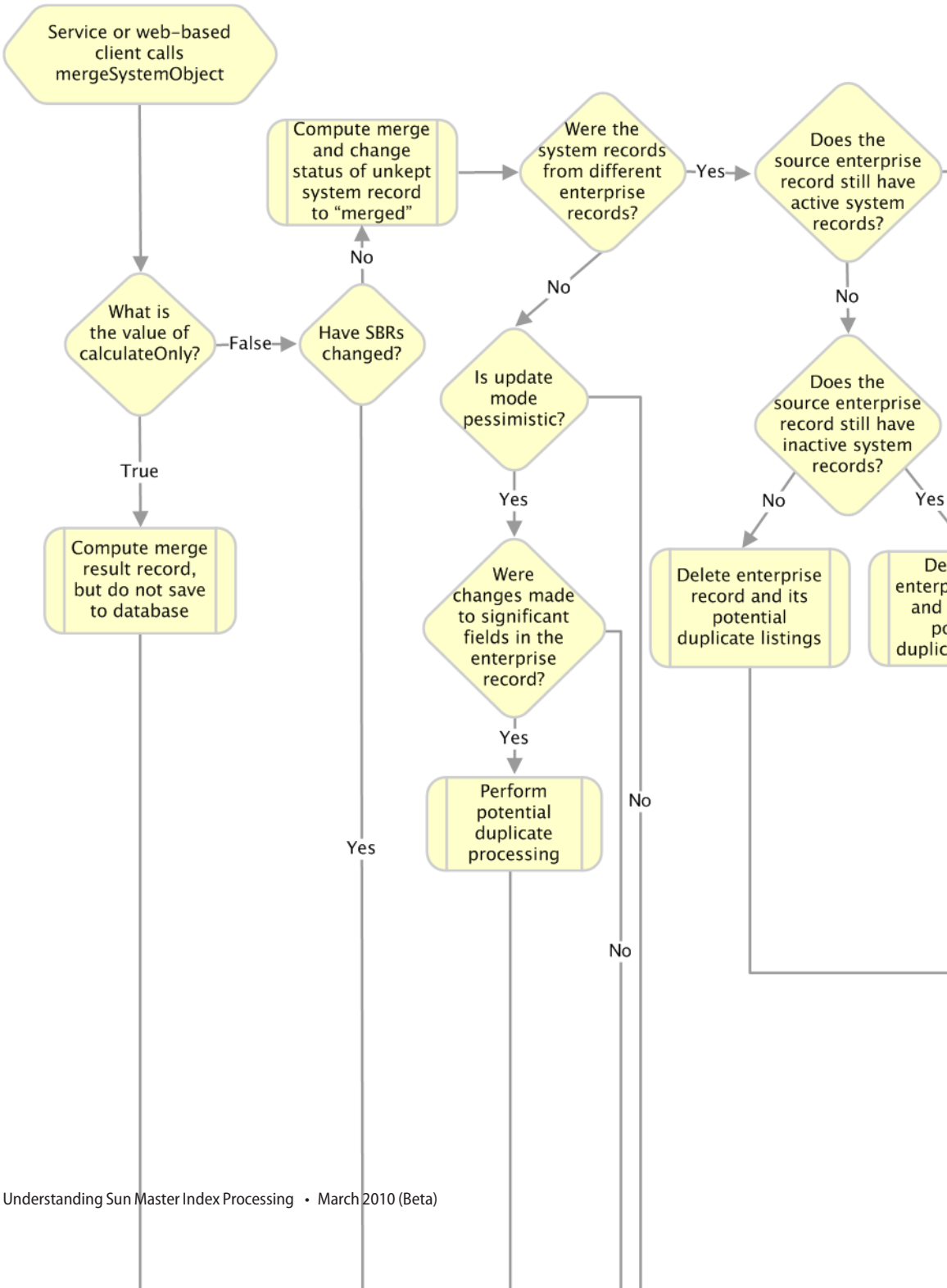
method is called with the *calculateOnly* parameter set to true in order to display the merge result record for you to view. When you confirm the merge, the MIDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed. Two of the merge methods compare the SBRs of the records with their corresponding SBRs in the database to ensure that no updates were made to the records before finalizing the merge.

When this method is called with *calculateOnly* set to “false”, the application changes the status of the merged system record to “merged”. If the system records were merged within the same enterprise record and the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record.

If the system records originated from two different enterprise records and the enterprise record that contained the unkept system record no longer has any active system records but does contain inactive system records, that enterprise record is deactivated and all associated potential duplicate listings are deleted. (Note that if the system records are unmerged, the enterprise record is reactivated.) If the enterprise record that contained the unkept system record no longer has any system records, that enterprise record is deleted along with any potential duplicate listings.

If both enterprise records are still active and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR for each enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record.

[Figure 13](#) illustrates the processing steps, and includes the check for SBR differences, which only occurs in two of the merge methods.



transferSystemObject

This transfers a system record from one enterprise record to another. This method is not called from the MIDM. If the enterprise record from which the system record was transferred no longer has any active system records (but still contains deactivated system records), that enterprise record is deactivated and any associated potential duplicate listings are removed. If the enterprise record from which the system record was transferred no longer has any system records, that enterprise record is deleted along with all associated potential duplicate listings. If both enterprise records are still active and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR for each enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record. [Figure 14](#) illustrates the processing steps.

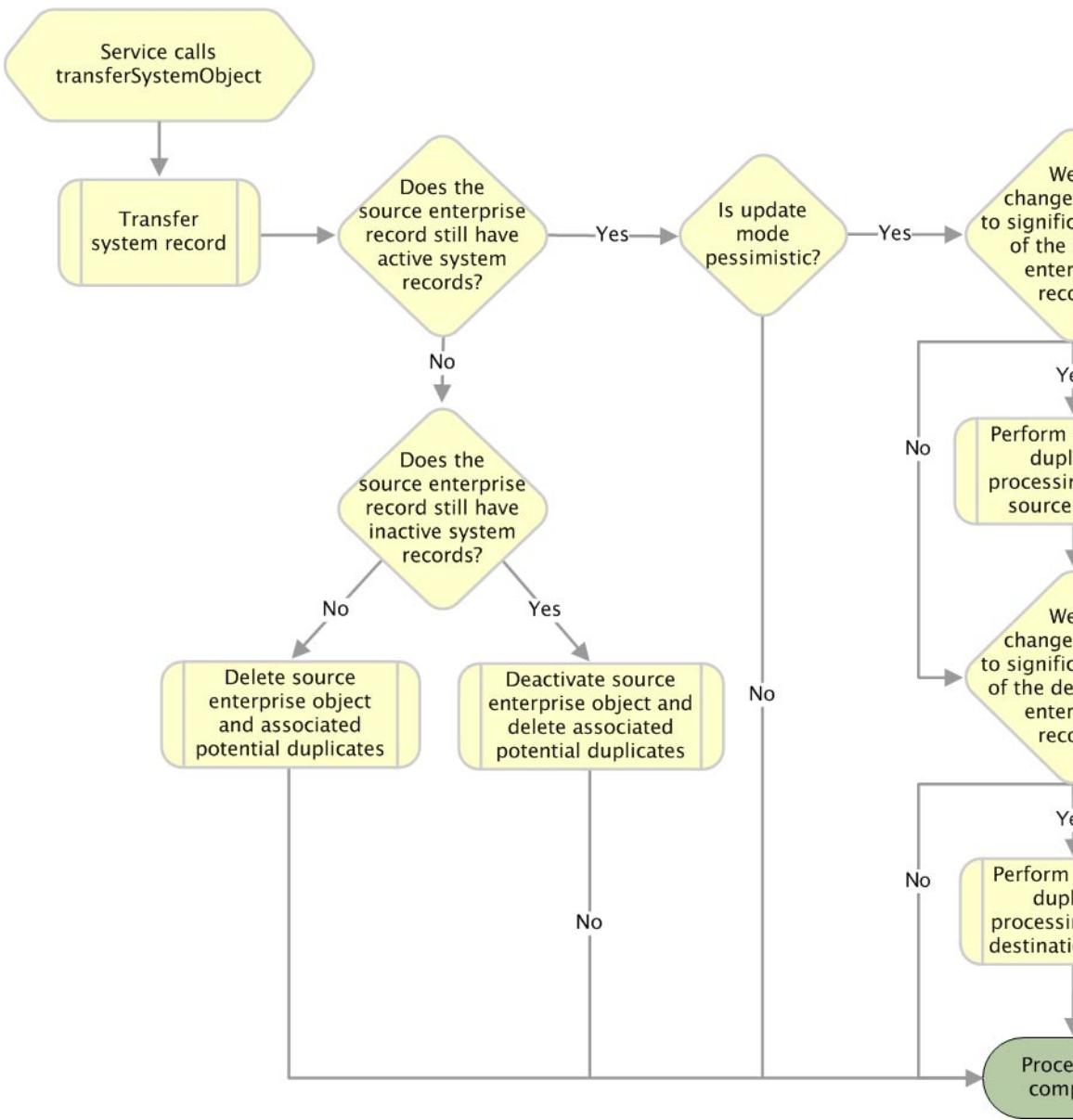


FIGURE 14 transferSystemObject Processing

undoAssumedMatch

This method reverses an assumed match made by the master index application, using the information from the system record that created the assumed match to create a new enterprise record. The MIDM calls this method when you confirm the transaction after selecting Undo Assumed Match. Potential duplicates are calculated for the new record regardless of the update mode. If the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the original enterprise record. If key fields were updated, potential duplicates are recalculated for the enterprise record. [Figure 15](#) illustrates the processing steps.

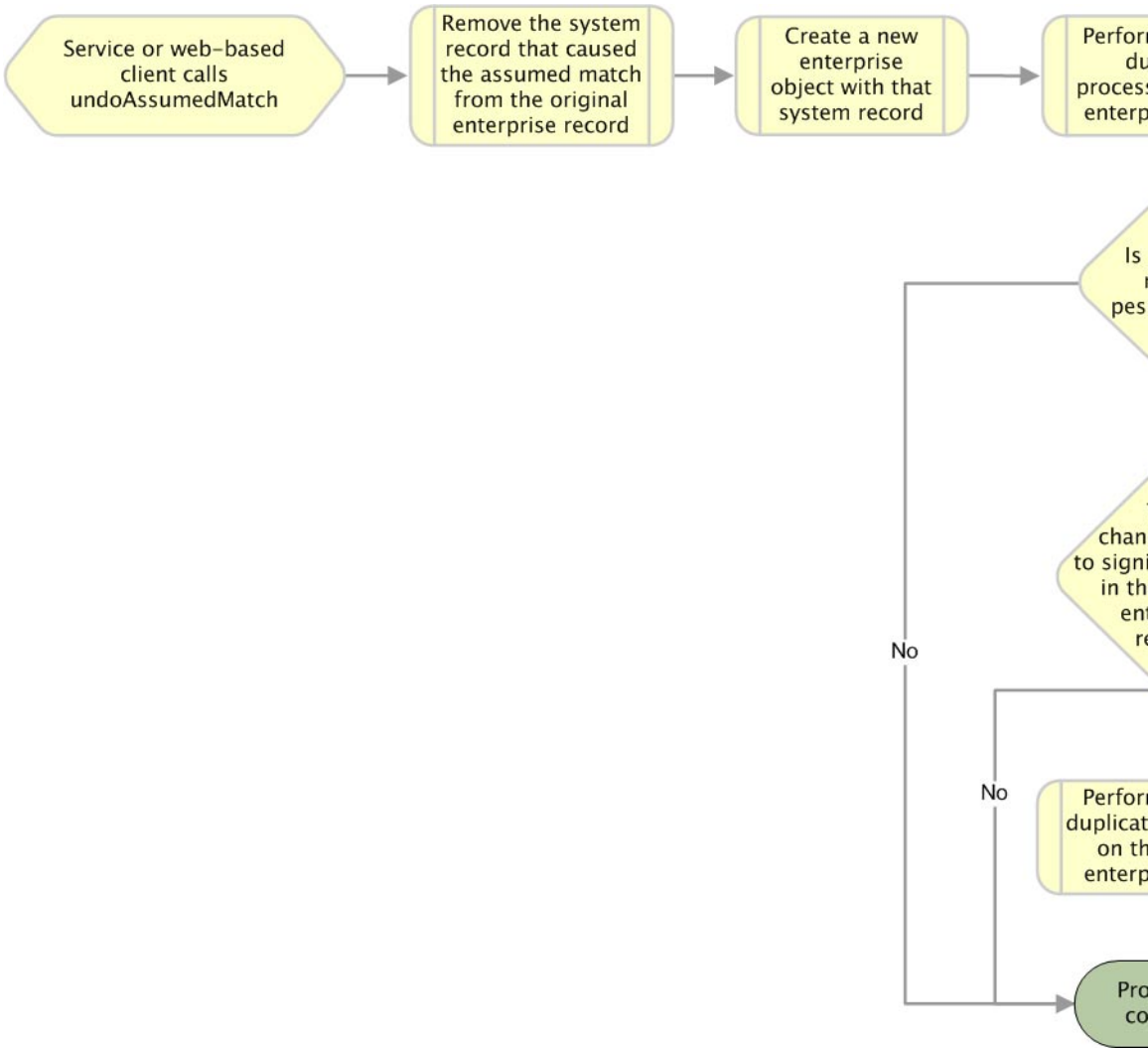


FIGURE 15 `undoAssumedMatch` Processing

`unmergeEnterpriseObject`

There are two methods that unmerge two enterprise records that were previously merged. One method unmerges the record without checking to make sure the SBR of the active record was not changed by another process before finalizing the merge and one method performs the SBR check (see the Javadocs provided with Sun Master Index for more information). The MIDM calls this method twice during an unmerge transaction. When you first click Unmerge, the method is called with the *calculateOnly* parameter set to true in order to display the unmerge

result records for you to view. When you confirm the unmerge, the MIDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged enterprise record back to active and recalculates potential duplicate listings for the record. If the update mode is set to “pessimistic”, the application checks whether any key fields were updated in the SBR of the enterprise record that was still active after the merge. If key fields were updated, potential duplicates are recalculated for that enterprise record. [Figure 16](#) illustrates the processing steps and includes the check for SBR updates.

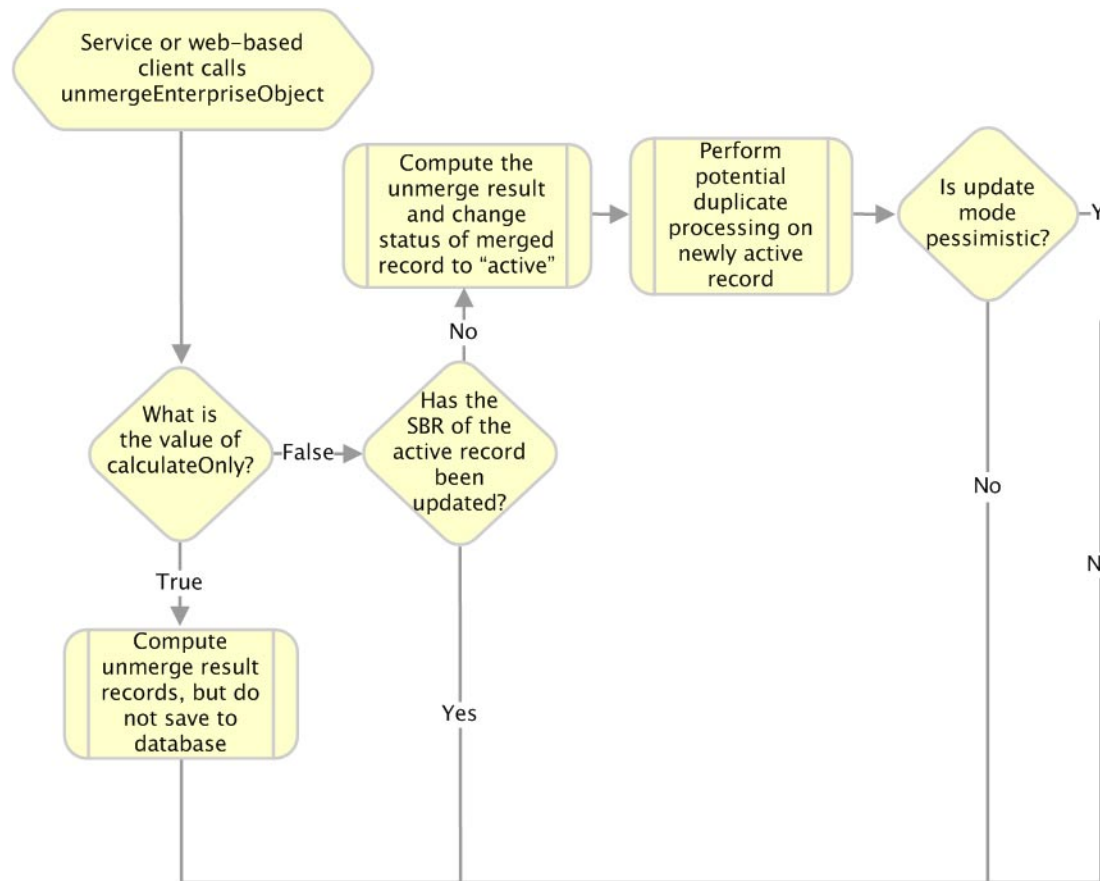


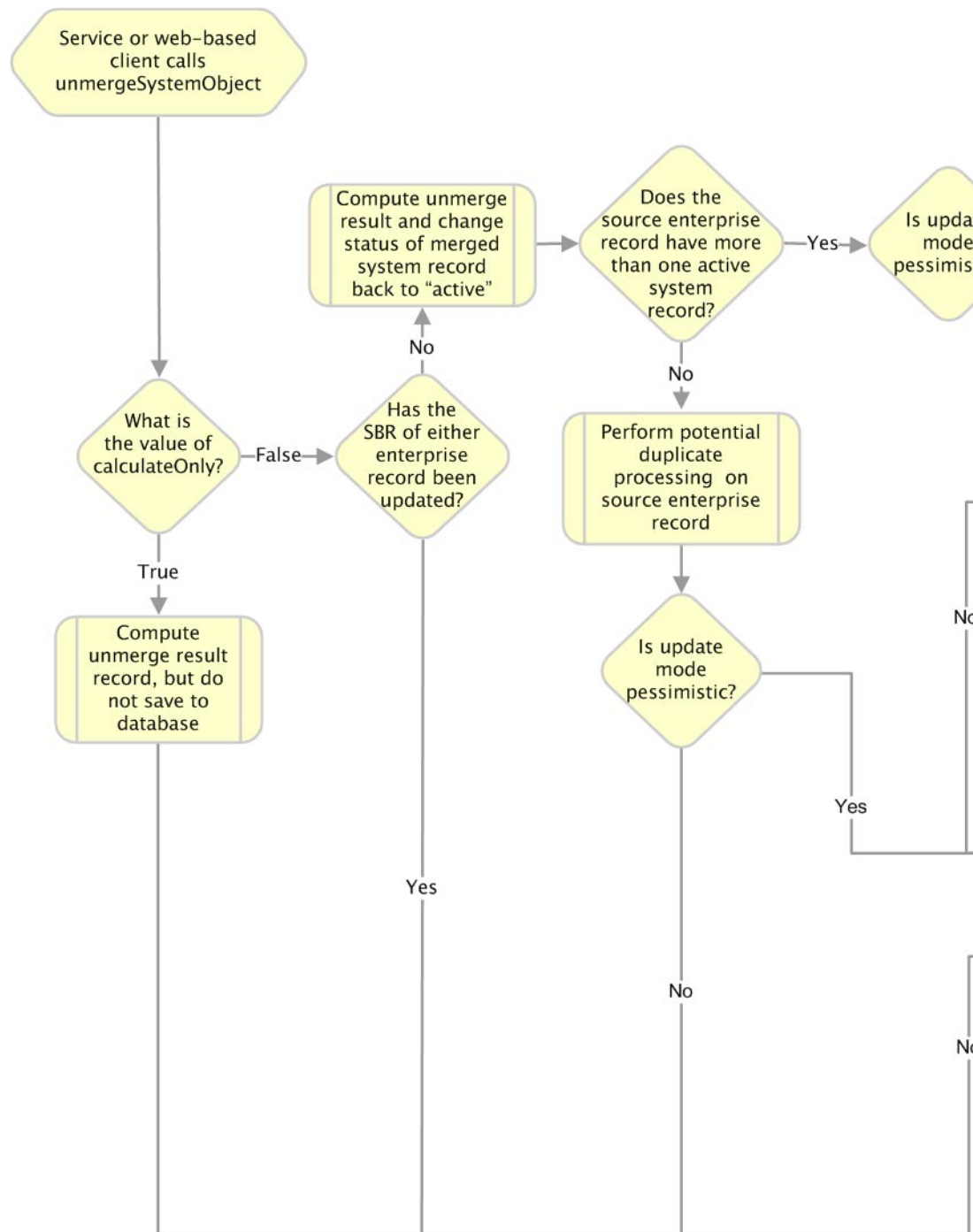
FIGURE 16 `unmergeEnterpriseObject` Processing

unmergeSystemObject

There are two methods that unmerge two system records that had previously been merged. One method unmerges the record without checking to make sure the SBR of the active record was not changed by another process before finalizing the merge and one method performs the SBR check (see the Javadocs provided with Sun Master Index for more information). The MIDM calls this method twice during a system record unmerge transaction. When you first click Unmerge, the method is called with the *calculateOnly* parameter set to true in order to display the unmerge result record for you to view. When you confirm the unmerge, the MIDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged system record back to active. If the source enterprise record (the record that contained the merge result system record after the merge) has more than one active system record after the unmerge and the update mode is set to pessimistic, the application checks whether any key fields were updated in that record. If key fields were updated, potential duplicates are recalculated for the source enterprise record.

If the source enterprise record has only one active system, potential duplicate processing is performed regardless of the update mode and of whether there were any changes to key fields. If the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR for destination enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record. [Figure 17](#) illustrates the processing steps, assuming the system record unmerge involves two enterprise records and including the check for SBR updates.



updateEnterpriseDupRecalc

This method updates the database to reflect new values for an enterprise record. It processes records in the same manner as `updateEnterpriseObject`, but provides an override flag for the update mode that allows you to defer potential duplicate processing. The MIDM does not call this method.

If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record was changed during the transaction but is still active and the *performPessimistic* parameter is set to true, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. Figure 18 illustrates the processing steps.

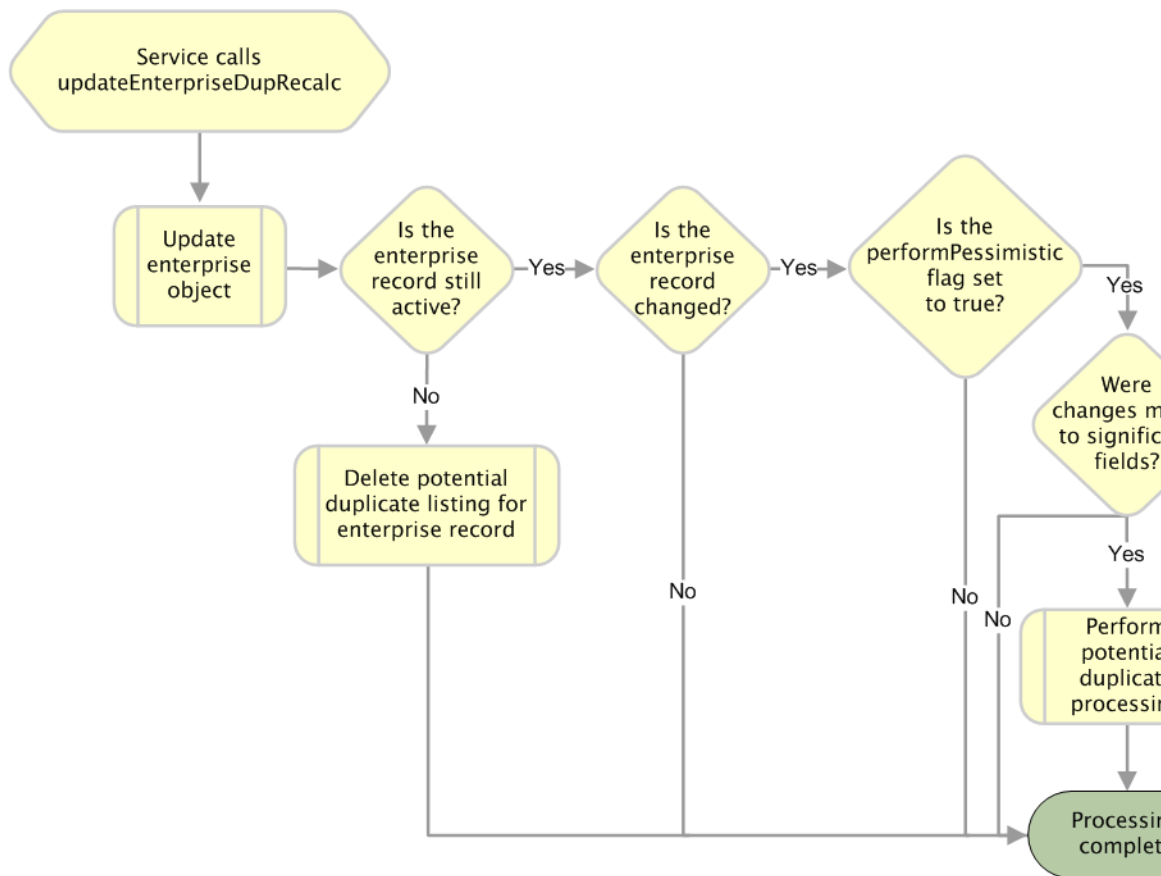


FIGURE 18 updateEnterpriseDupRecalc Processing

updateEnterpriseObject

This method updates the database to reflect new values for an enterprise record, and is called from the MIDM when you commit changes to an existing record. If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record is still active, was changed during the transaction, and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. [Figure 19](#) illustrates the processing steps.

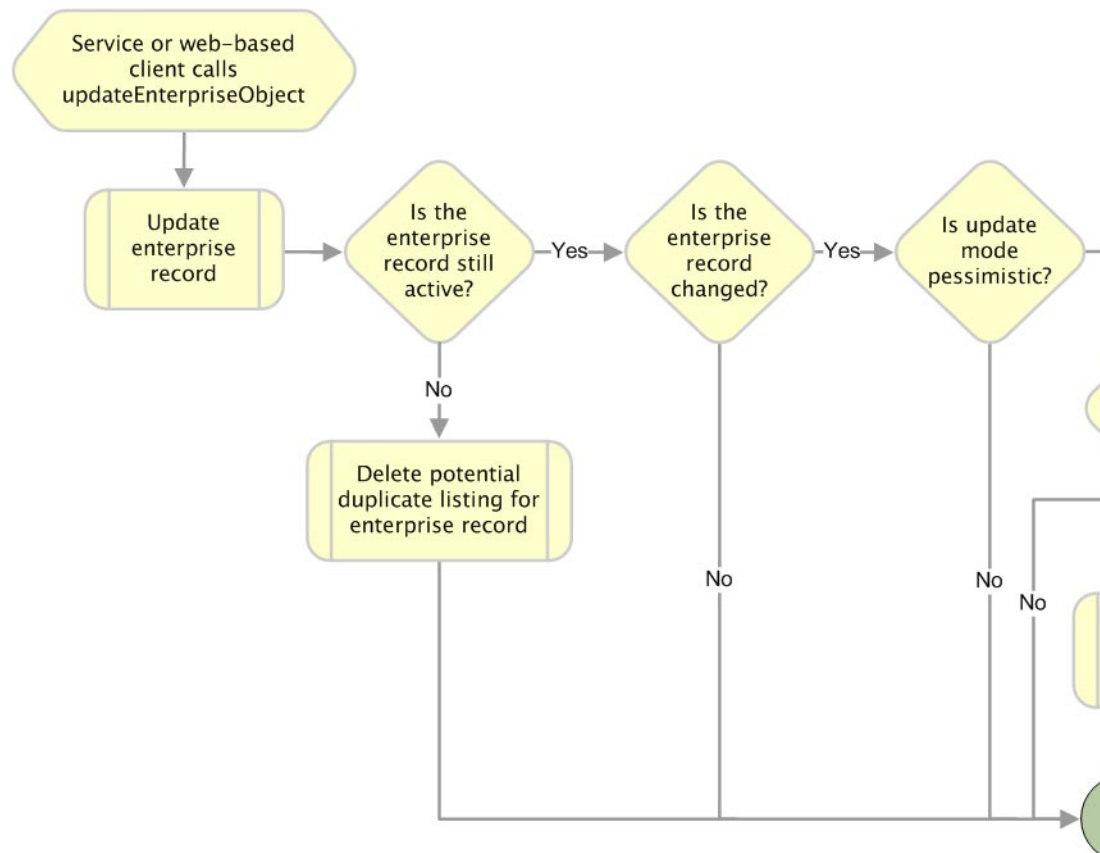


FIGURE 19 updateEnterpriseObject Processing

updateSystemObject

There are two methods that update the database to reflect new values for a system record. One method updates the record without checking that there were no concurrent changes to the record, and the other method compares the SBR of the associated enterprise object in the transaction with that in the database to be sure there were no concurrent changes (see the Javadocs provided with Sun Master Index for more information). The MIDM calls the method that checks for SBR changes when you commit changes to an existing system record.

If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record was changed during the transaction and is still active, and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. [Figure 20](#) illustrates the processing steps and includes the check for SBR changes though it only occurs with one of the methods.

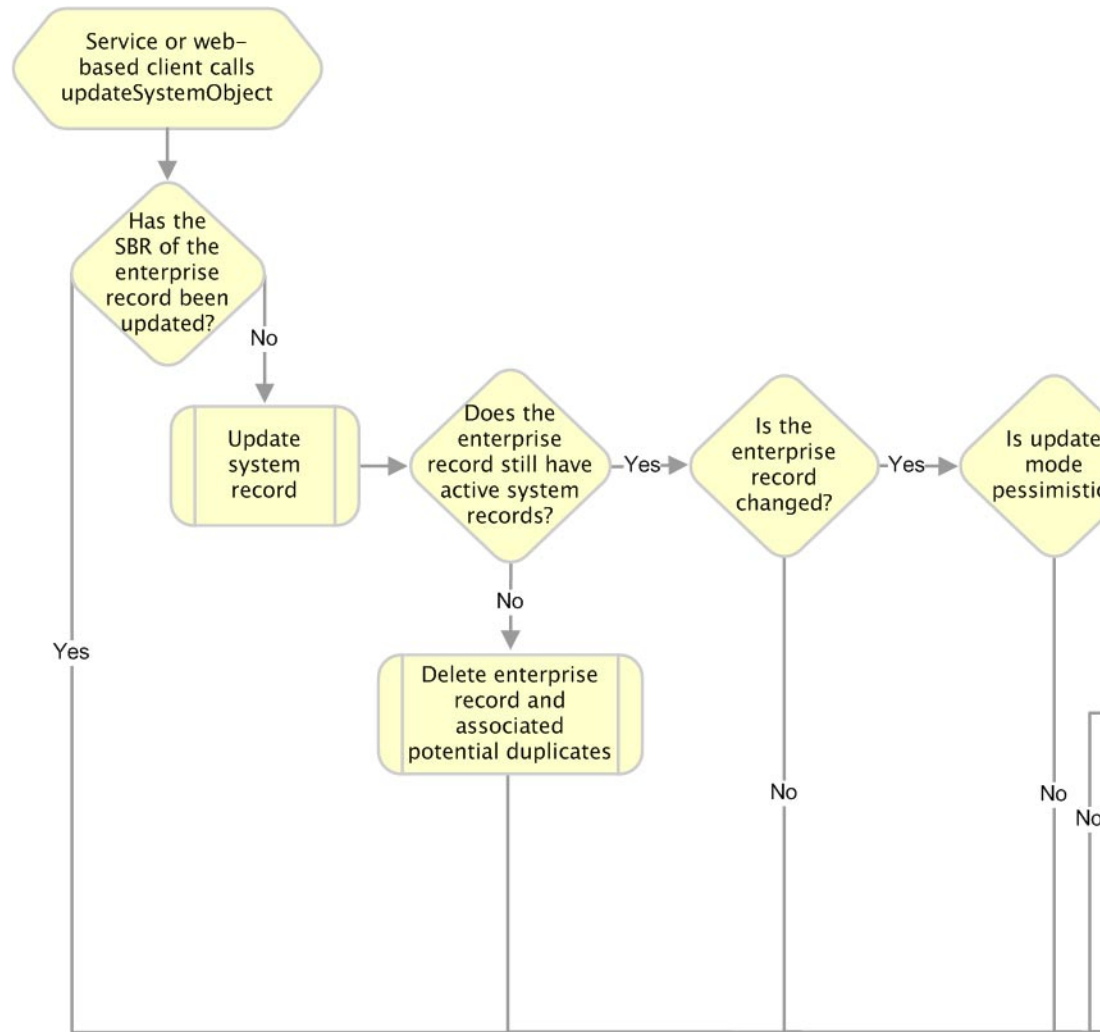


FIGURE 20 updateSystemObject Processing

The Database Structure

This chapter provides information about the master index database, including descriptions of each table and a sample entity relationship diagram. All information in this chapter pertains to the default version of the database. Your implementation might vary depending on the customization made to the master index object structure and to the scripts used to create the master index database. This chapter discusses the following topics:

- [“About the Database” on page 43](#)
- [“Database Table Details” on page 45](#)
- [“Sample Database Model” on page 61](#)

About the Database

The master index database stores information about the entities being indexed, such as people or businesses. The database stores records from local systems in their original form and also stores a record for each object that is considered to be the single best record (SBR).

The structure of the database tables that store object information is dependent on the information specified in the `object.xml` file created by the wizard. Sun Master Index generates a script to create the tables and fields in the database based on the information in `object.xml`. If you update `object.xml`, regenerating the application updates the database scripts accordingly. This allows you to define the database as you define the object structure.

While most of the structures created in the database are based on information in `object.xml`, some of the tables, such as `sbyn_seq_table` and `sbyn_common_detail`, are standard for all implementations. The database includes tables that store information about the objects defined for the master index application as well as tables that store common maintenance information, transactional information, and external system information. The database includes the tables listed in [Table 1](#).

TABLE 1 Master Index Database Tables

Table Name	Description
SBYN_OBJECT_NAME	Stores information for the parent objects associated with local system records. This database table is named by the parent object name. For example, a table storing company objects is named <code>sbyn_company</code> ; a table storing person objects is named <code>sbyn_person</code> . Only one table stores parent object information for system records.
SBYN_OBJECT_NAMESBR	Stores information for the parent objects associated with single best records. This database table is named by the parent object name followed by “SBR”. For example, a table storing company objects is named <code>sbyn_companysbr</code> ; a table storing person objects is named <code>sbyn_personsbr</code> . Only one table stores parent object information for SBRs.
SBYN_CHILD_OBJECT	Stores information for child objects associated with local system records. These database tables are named by their object name. For example, a table storing address objects is named <code>sbyn_address</code> ; a table storing comment objects is named <code>sbyn_comment</code> . One database table is created for each child object defined in the object structure.
SBYN_CHILD_OBJECTSBR	Stores information for child objects associated with a single best record. These database tables are named by their object name followed by “SBR”. For example, a table storing address objects is named <code>sbyn_addresssbr</code> ; a table storing comment objects is named <code>sbyn_commentsbr</code> . One SBR database table is created for each child object defined in the object structure.
SBYN_APPL	Lists the applications with which each item in <code>stc_common_header</code> is associated. Currently the only item in this table is eView .
SBYN_ASSUMEDMATCH	Stores information about records that were automatically matched by the master index application.
SBYN_AUDIT	Stores audit information about each time object information is accessed from the MIDM. Note – If audit logging is enabled, this table can grow very large and might require periodic archiving.
SBYN_COMMON_DETAIL	Contains all of the processing codes associated with the items listed in <code>sbyn_common_header</code> .
SBYN_COMMON_HEADER	Contains a list of the different types of processing codes used by the master index application. These types are also associated with the drop-down lists you can specify for the MIDM.
SBYN_ENTERPRISE	Stores the local ID and system pairs, along with their associated EUID.

TABLE 1 Master Index Database Tables (Continued)

Table Name	Description
SBYN_MERGE	Stores information about all merge and unmerge transactions processed from either external systems or the MIDM.
SBYN_OVERWRITE	Stores information about fields that are locked for updates in an SBR.
SBYN_POTENTIALDUPLICATES	Stores a list of potential duplicate records and flags potential duplicate pairs that have been resolved.
SBYN_SEQ_TABLE	Stores the sequential codes that are used in other tables in the database, such as EUIDs, transaction numbers, and so on.
SBYN_SYSTEMOBJECT	Stores information about the system objects in the database, including the local ID and system, create date and user, status, and so on.
SBYN_SYSTEMS	Stores a list of systems in your organization, along with defining information.
SBYN_SYSTEMSBR	Stores transaction information about an SBR, such as the create or update date, status, and so on.
SBYN_TRANSACTION	Stores a history of changes to each record stored in the database.
SBYN_USER_CODE	Like the sbyn_common_detail table, this table stores processing codes and drop-down list values. This table contains additional validation information that allows you to validate information in a dependent field (for example, to validate cities against the entered postal code).

Database Table Details

The tables on this and the following pages describe each column in the default database tables. The columns are identical for Oracle and SQL Server databases, but the data types differ in some cases. [Table 2](#) lists the data type differences.

TABLE 2 Oracle and SQL Server Data Type Differences

Oracle Data Type	SQL Server Data Type	MySQL Data Type
BLOB	Varbinary(MAX)	blob
DATE	DateTime	datetime
INTEGER	Int	integer
LONG	Varchar(MAX)	mediumtext
NUMBER	Numeric	decimal
TIMESTAMP	DateTime	datetime

TABLE 2 Oracle and SQL Server Data Type Differences (Continued)

Oracle Data Type	SQL Server Data Type	MySQL Data Type
VARCHAR2	Varchar	varchar

SBYN_OBJECT_NAME

This table stores the parent object in each system record received by the master index application. It is linked to the tables that store each child object in the system record by the *object_nameid* column (where *object_name* is the name of the parent object). This table contains the columns listed below regardless of the design of the object structure, and also contains a column for each field you defined for the parent object in *object.xml*. Columns to store standardized or phonetic versions of certain fields are automatically added when you specify certain match types in the wizard.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 3 SBYN_OBJECT_NAME Table Description

Column Name	Data Type	Column Description
SYSTEMCODE	VARCHAR2(20)	The system code for the system record.
LID	VARCHAR2(25)	A local identification code assigned by the specified system.
OBJECT_NAMEID	VARCHAR2(20)	A unique ID for the parent object in a system record. This is named according to the parent object. For example, if the parent object is “Company”, the name of this column is “companyid”; if the parent object is “Person”, the name of this column is “personid”.

SBYN_OBJECT_NAMESBR

This table stores the parent object of the SBR for each enterprise object in the master index database. It is linked to the tables that store each child object in the SBR by the *object_nameid* column (where *object_name* is the name of the parent object). This table contains the columns listed below regardless of the design of the object structure, and also contains a column for each field defined for the parent object in *object.xml*. In addition, columns to store standardized or phonetic versions of certain fields are automatically added when you specify certain match types in the wizard.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 4 SBYN_ OBJECT_NAMESBR Table Description

Column Name	Data Type	Column Description
EUID	VARCHAR2(20)	The enterprise unique identifier assigned by the master index application.
OBJECT_NAMEID	VARCHAR2(20)	A unique ID for the parent object in a system record. This is named according to the parent object. For example, if the parent object is “Company”, the name of this column is “companyid”; if the parent object is “Person”, the name of this column is “personid”.

SBYN_CHILD_OBJECT

The *sbyn_child_object* tables (where *child_object* is the name of a child object in the object structure) store information about the child objects associated with a system record in the master index application. All tables storing child object information for system records contain the columns listed below. The remaining columns are defined by the fields you specify for each child object in the object structure definition file, including any standardized or phonetic fields.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 5 SBYN_CHILD_OBJECT and SBYN_CHILD_OBJECTSBR Table Description

Column Name	Data Type	Column Description
OBJECT_NAMEID	VARCHAR2(20)	The unique ID for the parent object associated with the child object in the system record.
CHILD_OBJECTID	VARCHAR2(20)	The unique ID for each record in the table. This column cannot be null.

SBYN_CHILD_OBJECTSBR

The *sbyn_child_objectsbr* tables (where *child_object* is the name of a child object in the object structure) store information about the child objects associated with an SBR in the master index application. All tables storing child object information for SBRs contain the columns listed below. The remaining columns are defined by the fields you specify for each child object in *object.xml*, including any standardized or phonetic fields.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 6 SBYN_CHILD_OBJECT and SBYN_CHILD_OBJECTSBR Table Description

Column Name	Data Type	Column Description
OBJECT_NAMEID	VARCHAR2(20)	The unique ID for the parent object associated with the child object in the SBR.
CHILD_OBJECTID	VARCHAR2(20)	The unique ID for each record in the table. This column cannot be null.

SBYN_APPL

This table stores information about the applications used in the master index system. Currently, there is only one entry, “eView”. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 7 SBYN_APPL Table Description

Column Name	Data Type	Description
APPL_ID	NUMBER(10)	The unique sequence number code for the listed application.
CODE	VARCHAR2(8)	A unique code for the application.
DESCR	VARCHAR2(30)	A brief description of the application.
READ_ONLY	CHAR(1)	An indicator of whether the current entry can be modified. If the value of this column is “Y”, the entry cannot be modified.
CREATE_DATE	DATE	The date the application entry was created.
CREATE_USERID	VARCHAR2(20)	The logon ID of the user who created the application entry.

SBYN_ASSUMEDMATCH

This table maintains a record of each assumed match transaction that occurs in the master index application, allowing you to review these transactions and, if necessary, reverse an assumed match. This table can grow quite large over time; it is recommended that the table be archived periodically.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 8 SBYN_ASSUMEDMATCH Table Description

Column Name	Data Type	Description
ASSUMEDMATCHID	VARCHAR2(20)	The unique ID for the assumed match transaction.
EUID	VARCHAR2(20)	The EUID into which the incoming record was merged.
SYSTEMCODE	VARCHAR2(20)	The system code for the source system (that is, the system from which the incoming record originated).
LID	VARCHAR2(25)	The local ID of the record in the source system.
WEIGHT	VARCHAR2(20)	The matching weight between the incoming record and the EUID record into which it was merged.
TRANSACTION NUMBER	VARCHAR2(20)	The transaction number associated with the assumed match.

SBYN_AUDIT

This table maintains a log of each instance in which any of the master index database tables are accessed through the MIDM. This includes each time a record appears on a search results page, a comparison page, the Record Details page, and so on. This log is only maintained if the MIDM is configured for it. This table can grow very large over time and might require periodic archiving.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 9 SBYN_AUDIT Table Description

Column Name	Data Type	Description
AUDIT_ID	VARCHAR2(20)	The unique identification code for the audit record. This column cannot be null.
PRIMARY_OBJECT_TYPE	VARCHAR2(20)	The name of the parent object as defined in <code>object.xml</code> .
EUID	VARCHAR2(15)	The EUID whose information was accessed during an MIDM transaction.
EUID_AUX	VARCHAR2(15)	The second EUID whose information was accessed during an MIDM transaction. A second EUID appears when viewing information about merge and unmerge transactions, comparisons, and so on.

TABLE 9 SBYN_AUDIT Table Description *(Continued)*

Column Name	Data Type	Description
FUNCTION (Oracle) OPERATION (SQL Server)	VARCHAR2(32)	The type of transaction that caused the audit record to be written. This column cannot be null.
DETAIL	VARCHAR2(120)	A brief description of the transaction that caused the audit record to be written.
CREATE_DATE	DATE	The date the transaction that created the audit record was performed. This column cannot be null.
CREATE_BY	VARCHAR2(20)	The user ID of the person who performed the transaction that caused the audit log. This column cannot be null.

SBYN_COMMON_DETAIL

This table stores the processing codes and description for all of the common maintenance data elements. This is the detail table for `sbyn_common_header`. Each data element in `sbyn_common_detail` is associated with a data type in `sbyn_common_header` by the `common_header_id` column. None of the columns in this table can be null.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 10 SBYN_COMMON_DETAIL Table Description

Column Name	Data Type	Description
COMMON_DETAIL_ID	NUMBER(10)	The unique identification code of the common table data element.
COMMON_HEADER_ID	NUMBER(10)	The unique identification code of the common table data type associated with the data element (as stored in the <code>common_header_id</code> column of the <code>sbyn_common_header</code> table).
CODE	VARCHAR2(20)	The processing code for the common table data element.
DESCR	VARCHAR2(50)	A description of the common table data element.
READ_ONLY	CHAR(1)	An indicator of whether the common table data element can be modified.
CREATE_DATE	DATE	The date the data element record was created.
CREATE_USERID	VARCHAR2(20)	The user ID of the person who created the data element record.

SBN_COMMON_HEADER

This table stores a description of each type of common maintenance data and is the header table for sbyn_common_detail. Together, these tables store the processing codes and drop-down menu descriptions for each common table data type. For a person index, common table data types might include Religion, Language, Marital Status, and so on. For a business index, common table data types might include Address Type, Phone Type, and so on. None of the columns in this table can be null.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 11 SBN_COMMON_HEADER Table Description

Column Name	Data Type	Description
COMMON_HEADER_ID	VARCHAR2(10)	The unique identification code of the common table data type.
APPL_ID	VARCHAR2(10)	The application ID from sbyn_appl that corresponds to the application for which the common table data type is used.
CODE	VARCHAR2(8)	A unique processing code for the common table data type.
DESCR	VARCHAR2(50)	A description of the common table data type.
READ_ONLY	CHAR(1)	An indicator of whether an entry in the table is read-only (if this column is set to “Y”, the entry is read-only).
MAX_INPUT_LEN	NUMBER(10)	The maximum number of characters allowed in the code column for the common table data type.
TYP_TABLE_CODE	VARCHAR2(3)	This column is not currently used.
CREATE_DATE	DATE	The date the common table data type record was created.
CREATE_USERID	VARCHAR2(20)	The user ID of the person who created the common table data type record.

SBN_ENTERPRISE

This table stores a list of all the system and local ID pairs assigned to the enterprise records in the database, along with the associated EUID for each pair. This table is linked to sbyn_systemobject by the systemcode and lid columns, and is linked to sbyn_systemsbr by the euid column. This table maintains links between the SBR and its associated system objects. None of the columns in this table can be null.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 12 SBYN_ENTERPRISE Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20)	The processing code of the system associated with the local ID.
LID	VARCHAR2(25)	The local ID associated with the system and EUID.
EUID	VARCHAR2(20)	The EUID associated with the local ID and system.

SBYN_MERGE

This table maintains a record of each merge transaction that occurs in the master index application, both through the MIDM and from external systems. It also records any unmerges that occur. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 13 SBYN_MERGE Table Description

Column Name	Data Type	Description
MERGE_ID	VARCHAR2(20)	The unique, sequential identification code of merge record. This column cannot be null.
KEPT_EUID	VARCHAR2(20)	The EUID of the record that was retained after the merge transaction. This column cannot be null.
MERGED_EUID	VARCHAR2(20)	The EUID of the record that was not retained after the merge transaction.
MERGE_TRANSACTIONNUM	VARCHAR2(20)	The transaction number associated with the merge transaction. This column cannot be null.
UNMERGE_TRANSACTIONNUM	VARCHAR2(20)	The transaction number associated with the unmerge transaction.

SBYN_OVERWRITE

This table stores information about the fields that are locked for updates in the SBRs. It stores the EUID of the SBR, the ePath to the field, and the current locked value of the field. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 14 SBYN_OVERWRITE Table Description

Column Name	Data Type	Description
EUID	VARCHAR2(20)	The EUID of an SBR containing fields for which the overwrite lock is set.
PATH	VARCHAR2(200)	The ePath to a field that is locked in an SBR from the MIDM.
TYPE	VARCHAR2(20)	The data type of a field that is locked in an SBR.
INTEGERDATA	NUMBER(38)	The data that is locked for overwrite in an integer field.
BOOLEANDATA	NUMBER(38)	The data that is locked for overwrite in a boolean field.
STRINGDATA	VARCHAR2(200)	The data that is locked for overwrite in a string field.
BYTedata	CHAR(2)	The data that is locked for overwrite in a byte field.
LONGDATA	LONG	The data that is locked for overwrite in a long integer field.
DATEDATA	DATE	The data that is locked for overwrite in a date field.
FLOATDATA	NUMBER(38,4)	The data that is locked for overwrite in a floating decimal field.
TIMESTAMPDATA	DATE	The data that is locked for overwrite in a timestamp field.

SBYN_POTENTIALDUPLICATES

This table maintains a list of all records that are potential duplicates of one another. It also maintains a record of whether a potential duplicate pair has been resolved or permanently resolved. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 15 SBYN_POTENTIALDUPLICATES Table Description

Column Name	Data Type	Description
POTENTIALDUPLICATEID	VARCHAR2(20)	The unique identification number of the potential duplicate transaction.
WEIGHT	VARCHAR2(20)	The matching weight of the potential duplicate pair.
TYPE	VARCHAR2(15)	This column is reserved for future use.

TABLE 15 SBYN_POTENTIALDUPLICATES Table Description *(Continued)*

Column Name	Data Type	Description
DESCRIPTION	VARCHAR2(120)	A description of what caused the potential duplicate flag.
STATUS	VARCHAR2(15)	The status of the potential duplicate pair. The possible values are: <ul style="list-style-type: none">■ U – Unresolved■ R – Resolved■ A – Resolved permanently
HIGHMATCHFLAG	VARCHAR2(15)	This column is reserved for future use.
RESOLVEDUSER	VARCHAR2(30)	The user ID of the person who resolved the potential duplicate status.
RESOLVEDDATE	DATE	The date the potential duplicate status was resolved.
RESOLVEDCOMMENT	VARCHAR2(120)	Comments regarding the resolution of the duplicate status. This is not currently used.
EUID2	VARCHAR2(20)	The EUID of the second record in the potential duplicate pair.
TRANSACTIONNUMBER	VARCHAR2(20)	The transaction number associated with the transaction that produced the potential duplicate flag.
EUID1	VARCHAR2(20)	The EUID of the first record in the potential duplicate pair.

SBYN_SEQ_TABLE

This table controls and maintains a record of the sequential identification numbers used in various tables in the database, ensuring that each number is unique and assigned in order. Several of the ID numbers maintained in this table are determined by the object structure. The numbers are assigned sequentially, but are cached in chunks of 1000 numbers for optimization (so the application does not need to query the `sbyn_seq_table` table for each transaction). The chunk size for the EUID sequence is configurable. If the Repository server is reset before all allocated numbers are used, the unused numbers are discarded and never used, and numbering is restarted at the beginning of the next 1000-number chunk.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 16 SBYN_SEQ_TABLE Table Description

Column Name	Data Type	Description
SEQ_NAME	VARCHAR2(20)	The name of the object for which the sequential ID is stored.
SEQ_COUNT	NUMBER(38)	The current value of the sequence. The next record will be assigned the current value plus one.

The default sequence numbers are listed in the following table.

TABLE 17 Default Sequence Numbers

Sequence Name	Description
EUID	The sequence number that determines how EUIDs are assigned to new records. The chunk size for the EUID sequence number is configurable in the <code>master.xml</code> file.
POTENTIALDUPLICATE	The sequence number assigned each potential duplicate transaction record in <code>sbyn_potentialduplicates</code> (column name “potentialduplicateid”).
TRANSACTIONNUMBER	The sequence number assigned to each transaction in the master index application. This number is stored in <code>sbyn_transaction</code> (column name “transactionnumber”).
ASSUMEDMATCH	The sequence number assigned to each assumed match transaction record in <code>sbyn_assumedmatch</code> (column name “assumedmatchid”).
AUDIT	The sequence number assigned to each audit log record in <code>sbyn_audit</code> (column name “audit_id”).
MERGE	The sequence number assigned to each merge transaction in <code>sbyn_merge</code> (column name “merge_id”).
SBYN_APPL	The sequence number assigned to each application listed in <code>sbyn_appl</code> (column name “appl_id”).
SBYN_COMMON_HEADER	The sequence number assigned to each common table data type listed in <code>sbyn_common_header</code> (column name “common_header_id”).
SBYN_COMMON_DETAIL	The sequence number assigned to each common table data element listed in <code>sbyn_common_detail</code> (column name “common_detail_id”).
<i>OBJECT_NAME</i>	Each parent and child object system record table is assigned a sequential ID. The column names are named after the object (for example, <code>sbyn_address</code> has a sequential column named “addressid”). The parent object ID is included in each child object table.

TABLE 17 Default Sequence Numbers (Continued)

Sequence Name	Description
OBJECT_NAMESBR	Each parent and child object SBR table is assigned a sequential ID. The column names are named after the object (for example, sbyn_addresssbr has a sequential column named “addressid”). The parent object ID is included in each child object SBR table.

SBYN_SYSTEMOBJECT

This table stores information about the system records in the database, including their local ID and source system pairs. It also stores transactional information, such as the create or update date and function. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 18 SBYN_SYSTEMOBJECT Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20)	The processing code of the system associated with the local ID. This column cannot be null.
LID	VARCHAR2(25)	The local ID associated with the system and EUID (the associated EUID is found in sbyn_enterprise). This column cannot be null.
CHILDTYPE	VARCHAR2(20)	The type of object being processed (currently only the name of the parent object). This column is reserved for future use.
CREATEUSER	VARCHAR2(30)	The user ID of the person who created the system record.
CREATEFUNCTION	VARCHAR2(20)	The type of transaction that created the system record.
CREATEDATE	DATE	The date the system record was created.
UPDATEUSER	VARCHAR2(30)	The user ID of the person who last updated the system record.
UPDATEFUNCTION	VARCHAR2(20)	The type of transaction that last updated the system record.
UPDATEDATE	DATE	The date the system record was last updated.

TABLE 18 SBYN_SYSTEMOBJECT Table Description (Continued)

Column Name	Data Type	Description
STATUS	VARCHAR2(15)	The status of the system record. The status can be one of these values: <ul style="list-style-type: none"> ■ active ■ inactive ■ merged

SBYN_SYSTEMS

This table stores information about each system integrated into the master index environment, including the system’s processing code and name, a brief description, the format of the local IDs, and whether any of the system information should be masked. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 19 SBYN_SYSTEMS Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20)	The unique processing code of the system.
DESCRIPTION	VARCHAR2(120)	A brief description of the system, or the system name. This is the value that appears in the tree view panes of the MIDM for each system and local ID pair.
STATUS	CHAR(1)	The status of the system in the master index application. “A” indicates active and “D” indicates deactivated.
ID_LENGTH	NUMBER	The length of the local identifiers assigned by the system. This length does not include any additional characters added by the input mask.
FORMAT	VARCHAR2(60)	The required data pattern for the local IDs assigned by the system. For more information about possible values and using Java patterns, see “Patterns” in the class list for <code>java.util.regex</code> in the Javadocs provided with the Java™ 2 Platform, Standard Edition (J2SE™ platform). Note that the data pattern is also limited by the input mask described below. All regex patterns are supported if there is no input mask.

TABLE 19 SBYN_SYSTEMS Table Description (Continued)

Column Name	Data Type	Description
INPUT_MASK	VARCHAR2(60)	A mask used by the MIDM to add punctuation to the local ID. For example, the input mask DD-DDD-DDD inserts a hyphen after the second and fifth characters in an 8-digit ID. These character types can be used. <ul style="list-style-type: none">■ D – Numeric character■ L – Alphabetic character■ A – Alphanumeric character
VALUE_MASK	VARCHAR2(60)	A mask used to strip any extra characters that were added by the input mask for database storage. The value mask is the same as the input mask, but with an “x” in place of each punctuation mark. Using the input mask described above, the value mask is DDxDDDxDDD . This strips the hyphens before storing the ID.
CREATE_DATE	DATE	The date the system information was inserted into the database.
CREATE_USERID	VARCHAR2(20)	The logon ID of the user who inserted the system information into the database.
UPDATE_DATE	DATE	The most recent date the system’s information was updated.
UPDATE_USERID	VARCHAR2(20)	The logon ID of the user who last updated the system’s information.

SBYN_SYSTEMSBR

This table stores transactional information about the system records for the SBR, such as the create or update date and function. The `sbyn_systemsbr` table is indirectly linked to the `sbyn_systemobjects` table through `sbyn_enterprise`. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 20 SBYN_SYSTEMSBR Table Description

Column Name	Data Type	Description
EUID	VARCHAR2(20)	The EUID associated with system record (the associated system and local ID are found in <code>sbyn_enterprise</code>). This column cannot be null.

TABLE 20 SBYN_SYSTEMSBR Table Description *(Continued)*

Column Name	Data Type	Description
CHILDTYPE	VARCHAR2(20)	The type of object being processed (currently only the name of the parent object). This column is reserved for future use.
CREATESYSTEM	VARCHAR2(20)	The system in which the system record was created.
CREATEUSER	VARCHAR2(30)	The user ID of the person who created the system record.
CREATEFUNCTION	VARCHAR2(20)	The type of transaction that created the system record.
CREATEDATE	DATE	The date the system object was created.
UPDATEUSER	VARCHAR2(30)	The user ID of the person who last updated the system record.
UPDATEFUNCTION	VARCHAR2(20)	The type of transaction that last updated the system record.
UPDATEDATE	DATE	The date the system object was last updated.
STATUS	VARCHAR2(15)	The status of the enterprise record. The status can be one of these values: <ul style="list-style-type: none"> ■ active ■ inactive ■ merged
REVISIONNUMBER	NUMBER(38)	The revision number of the SBR. This is used for version control.

SBYN_TRANSACTION

This table stores a history of changes made to each record in the master index application, allowing you to view a transaction history and to undo certain actions, such as merging two object profiles. The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 21 SBYN_TRANSACTION Table Description

Column Name	Data Type	Description
TRANSACTIONNUMBER	VARCHAR2(20)	The unique number of the transaction.
LID1	VARCHAR2(25)	This column is reserved for future use.
LID2	VARCHAR2(25)	The local ID of the second system record involved in the transaction.

TABLE 21 SBYN_TRANSACTION Table Description (Continued)

Column Name	Data Type	Description
EUID1	VARCHAR2(20)	This column is reserved for future use.
EUID2	VARCHAR2(20)	The EUID of the second object profile involved in the transaction.
FUNCTION (Oracle) OPERATION (SQL Server)	VARCHAR2(20)	The type of transaction that occurred, such as update, add, merge, and so on.
SYSTEMUSER	VARCHAR2(30)	The logon ID of the user who performed the transaction.
TIMESTAMP	TIMESTAMP	The date and time the transaction occurred.
DELTA	BLOB	A list of the changes that occurred to system records as a result of the transaction.
SYSTEMCODE	VARCHAR2(20)	The processing code of the source system in which the transaction originated.
LID	VARCHAR2(25)	The local ID of the system record involved in the transaction.
EUID	VARCHAR2(20)	The EUID of the enterprise record involved in the transaction.

SBYN_USER_CODE

This table is similar to the `sbyn_common_header` and `sbyn_common_detail` tables in that it stores processing codes and drop-down list values. This table is used when the value of one field is dependent on the value of another. For example, if you store credit card information, you could list each credit card type and specify a required format for the credit card number field. The data stored in this table includes the processing code, a brief description, and the format of the dependent fields.

The following table lists Oracle data types. For information about the differences in data types between database vendors, see [Table 3](#).

TABLE 22 SBYN_USER_CODE Table Description

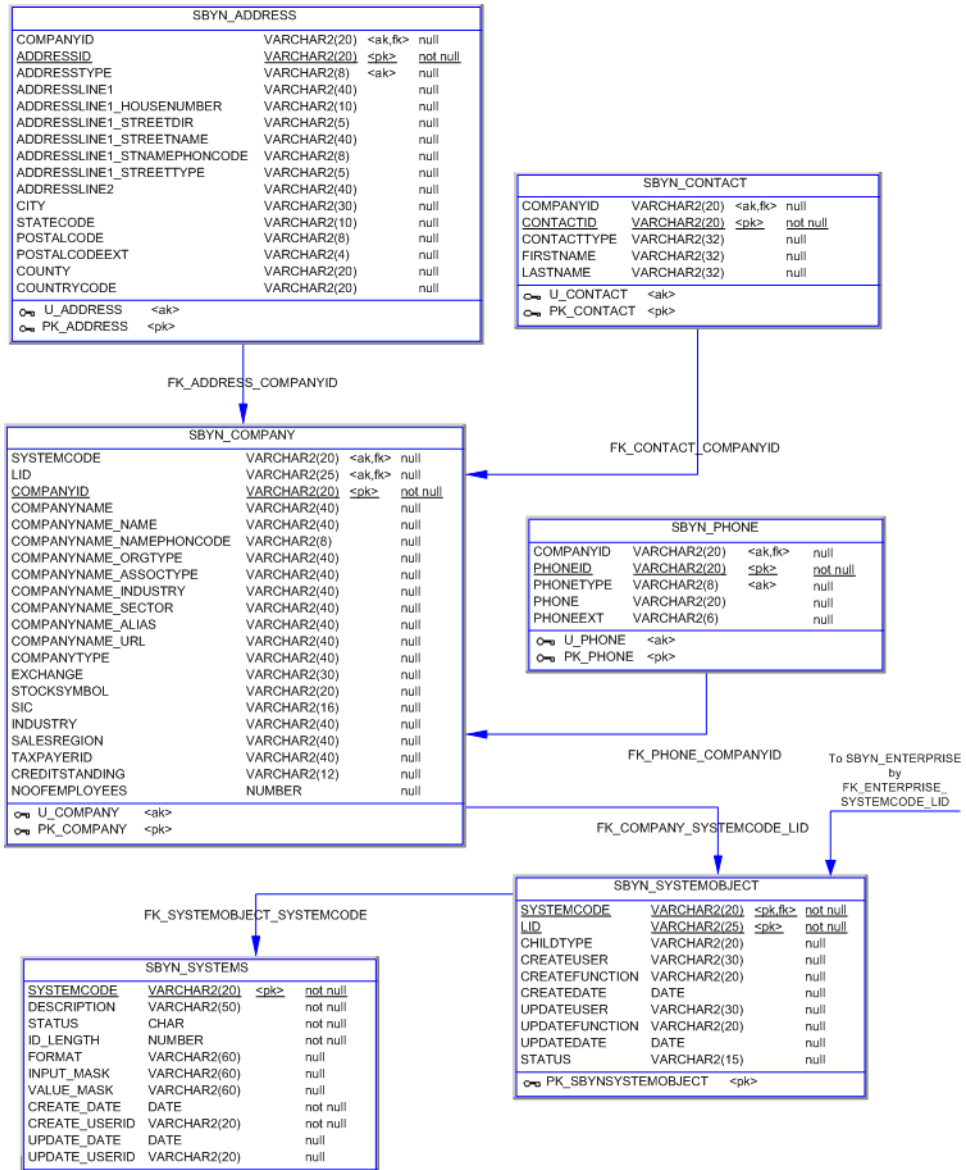
Column Name	Data Type	Description
CODE_LIST	VARCHAR2(20)	The code list name of the user code type (using the credit card example above, this might be similar to “CREDCARD”). This column links the values for each list.
CODE	VARCHAR2(20)	The processing code of each user code element.

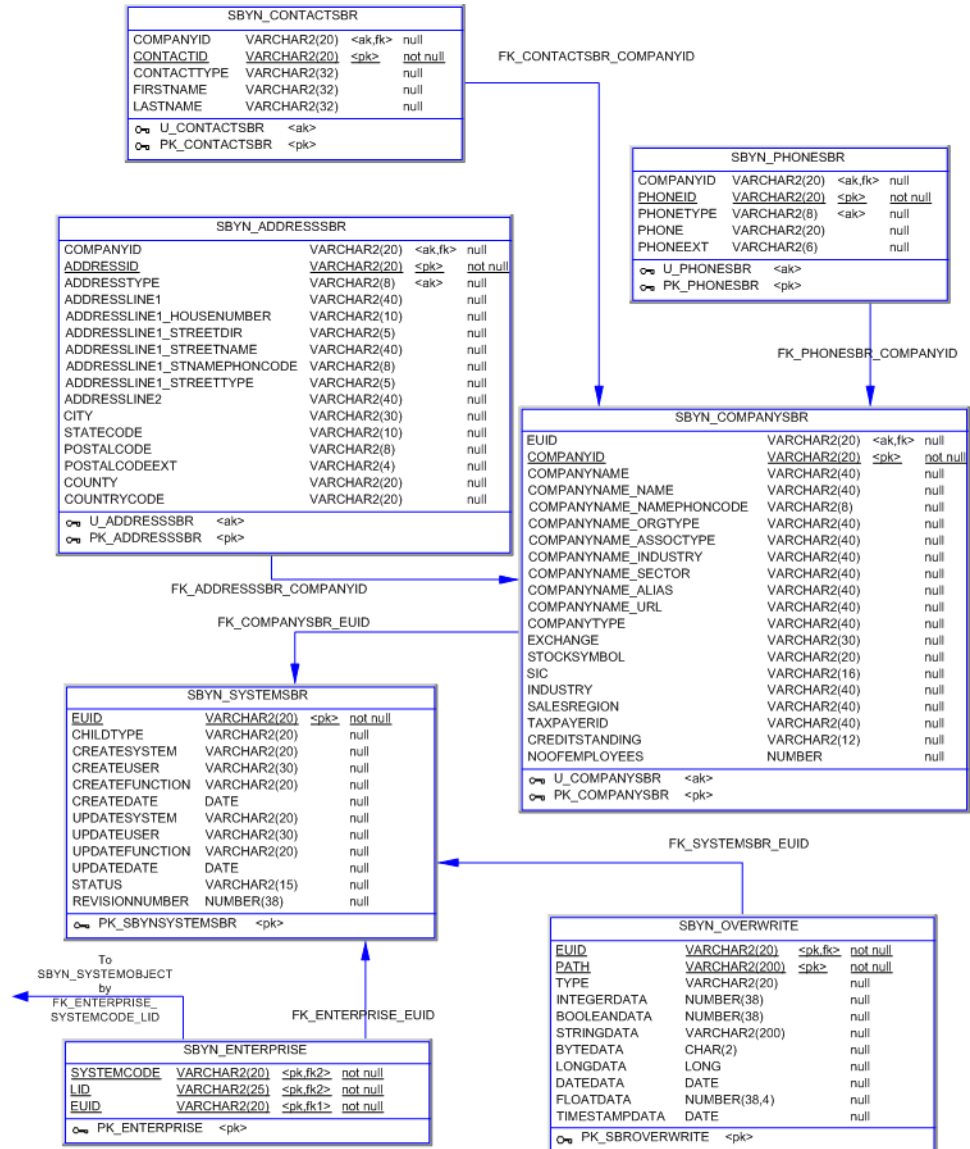
TABLE 22 SBYN_USER_CODE Table Description (Continued)

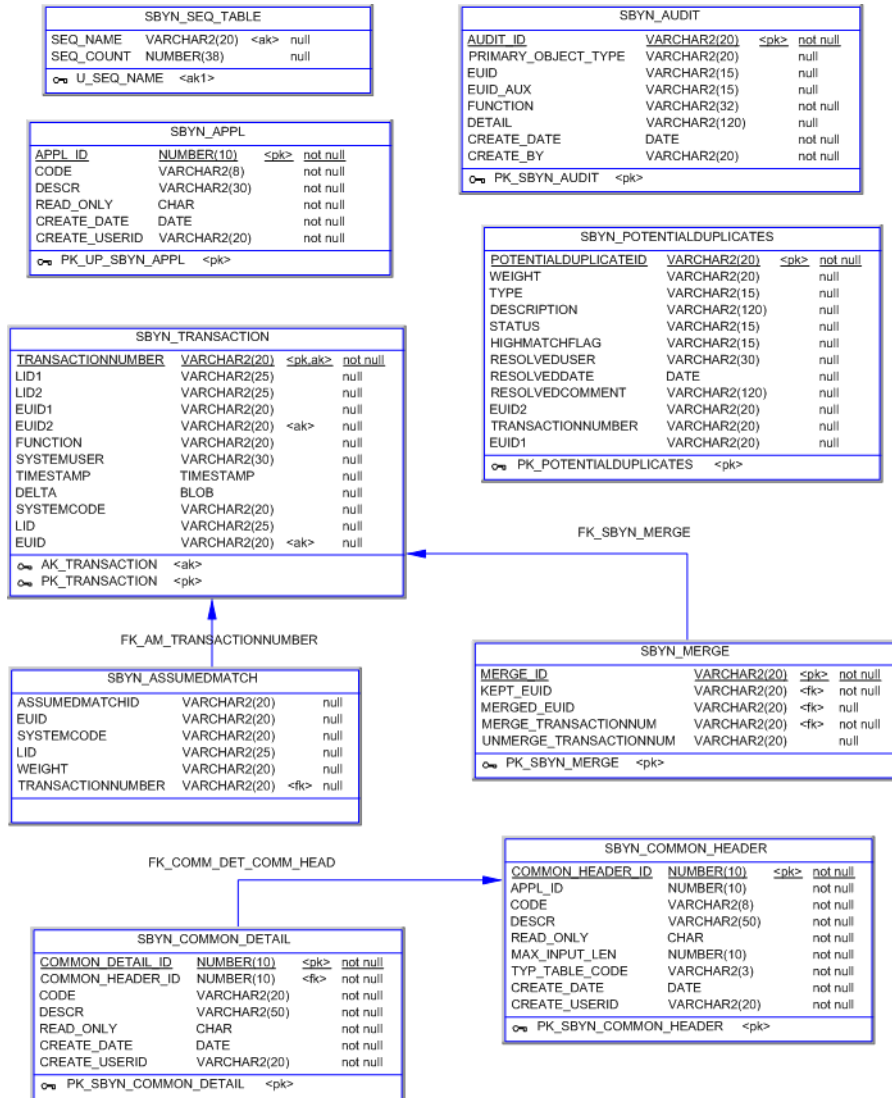
Column Name	Data Type	Description
DESCRIPTION	VARCHAR2(50)	A brief description or name for the user code. This is the value that appears in the drop-down list.
FORMAT	VARCHAR2(60)	The required data pattern for the field that is constrained by the user code. For more information about possible values and using Java patterns, see “Patterns” in the class list for <code>java.util.regex</code> in the Javadocs provided with the J2SE platform. Note that the data pattern is also limited by the input mask described below. All regex patterns are supported if there is no input mask.
INPUT_MASK	VARCHAR2(60)	A mask used by the MIDM to add punctuation to the constrained field. For example, the input mask DD-DDD-DDD inserts a hyphen after the second and fifth characters in an 8-digit ID. These character types can be used. <ul style="list-style-type: none"> ■ D – Numeric character ■ L – Alphabetic character ■ A – Alphanumeric character
VALUE_MASK	VARCHAR2(60)	A mask used to strip any extra characters that were added by the input mask for database storage. The value mask is the same as the input mask, but with an “x” in place of each punctuation mark. Using the input mask described above, the value mask is DDxDDDxDDD . This strips the hyphens before storing the ID.

Sample Database Model

The diagrams on the following pages illustrate the table structure and relationships for a sample master index Oracle database designed for storing information about companies. The diagrams display attributes for each database column, such as the field name, data type, whether the field can be null, and primary keys. They also show directional relationships between tables and the keys by which the tables are related. This diagram is very similar to SQL Server, with the exception of a few column name changes and some different data types as noted in the tables above.







Working with the Java API

Sun Master Index provides several Java classes and methods to use to transform and process data in a master index Project. The master index API is specifically designed to help you maintain the integrity of the data in the database by providing specific methods for updating, adding, and merging records in the database. This chapter discusses the following topics:

- “Overview” on page 65
- “Java Class Types” on page 65
- “Dynamic Web Service Methods” on page 66
- “Web Services Helper Classes” on page 85

Overview

This chapter provides an overview of the Java API for a Sun master index application, and describes the dynamic classes and methods that are generated based on the object structure of the master index application. For detailed information about the static classes and methods, refer to the Sun Master Index Javadocs, available at <http://developers.sun.com/docs/javacaps/api/javadocs/SunMasterIndex/index.html>. Unless otherwise noted, all classes and methods described in this chapter are public. Methods inherited from classes other than those described in this chapter are listed, but not described.

Java Class Types

Sun Master Index provides a set of static API classes that can be used with any object structure and any Sun master index application. Sun Master Index also generates several dynamic API classes that are specific to the object structure of each master index application. The dynamic classes contain similar methods, but the number and names of methods change depending on the object structure.

Static Classes

Static classes provide the methods you need to perform basic data cleansing and processing functions against incoming data, such as performing searches, reviewing potential duplicates, adding and updating records, and merging and unmerging records. The primary class containing these functions is the `MasterController` class, which includes the `executeMatch` methods. Several classes support the `MasterController` class by defining additional objects and functions. Documentation for the static methods is provided in Javadoc format.

Dynamic Web Services Classes

When you generate a master index Project, several dynamic methods are created in the Master Index EJB project that are specific to the object structure defined for the master index application. This includes classes that define each parent and child object in the object structure and that allow you to work with the data in each object. If the object structure is modified, regenerating the Project updates the dynamic methods for the new structure.

The EJB class that defines the primary web service methods is named after the parent object, and defines the main methods you use to transform and manipulate data between external systems and the master index database. There are also several helper classes that define the objects that are used as parameters and results for the primary web services, and that define the objects involved in Master Index processing.

The dynamic web services classes are located in the Master Index EJB project in the `com.sun.mdm.index.webservice` package under Source Packages. You can view the primary EJB class under Web Services in the EJB project. Additional supporting API classes are generated in the remaining Java packages under Source Packages.

Dynamic Web Service Methods

A set of Java methods are created in the Master Index EJB project for use in the client applications that call Master Index functions. These methods wrap static Java API methods, allowing them to work with the dynamic object classes. Many of these methods return objects of the dynamic object type, or they use these objects as parameters. In the following methods described for web services, *ObjectName* indicates the name of the parent object.

- “[activateEnterpriseRecord](#)” on page 67
- “[activateSystemRecord](#)” on page 68
- “[addSystemRecord](#)” on page 68
- “[deactivateEnterpriseRecord](#)” on page 69
- “[getSystemRecord](#)” on page 76
- “[getSystemRecordsByEUID](#)” on page 77
- “[getSystemRecordsByEUIDStatus](#)” on page 77
- “[lookupLIDs](#)” on page 78

- “deactivateSystemRecord” on page 70
- “executeMatch” on page 70
- “executeMatchUpdate” on page 71
- “getEnterpriseRecordByEUID” on page 72
- “getEnterpriseRecordByLID” on page 73
- “getEUID” on page 74
- “getLIDs” on page 74
- “getLIDsByStatus” on page 75
- “getSBR” on page 75
- “lookupPotentialDuplicates” on page 79
- “mergeEnterpriseRecord” on page 79
- “mergeSystemRecord” on page 80
- “searchBlock” on page 81
- “searchExact” on page 82
- “searchPhonetic” on page 82
- “transferSystemRecord” on page 83
- “updateEnterpriseRecord” on page 84
- “updateSystemRecord” on page 84

Dynamic API Method Descriptions

activateEnterpriseRecord

Description

This method changes the status of a deactivated enterprise object back to active.

Syntax

```
void activateEnterpriseRecord(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to activate.

Returns

None.

Throws

- ProcessingException
- UserException

activateSystemRecord

Description

This method changes the status of a deactivated system object back to active.

Syntax

```
void activateSystemRecord(String systemCode, String localId)
```

Parameters

Name	Type	Description
systemCode	String	The processing code of the system associated with the system record to be activated.
localID	String	The local identifier associated with the system record to be activated.

Returns

None.

Throws

- ProcessingException
- UserException

addSystemRecord

Description

This method adds the system object to the enterprise object associated with the specified EUID.

Syntax

```
void addSystemRecord(String euid, SystemObjectName sysObjBean)
```

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to which you want to add the system object.
sysObjBean	<i>SystemObjectName</i>	The Bean for the system object to be added to the enterprise object.

Returns

None.

Throws

- `ProcessingException`
- `UserException`

deactivateEnterpriseRecord

Description

This method changes the status of an active enterprise object to inactive.

Syntax

```
void deactivateEnterpriseRecord(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to deactivate.

Returns

None.

Throws

- `ProcessingException`
- `UserException`

deactivateSystemRecord

Description

This method changes the status of an active system object to inactive.

Syntax

```
void deactivateSystemRecord(String systemCode, String localId)
```

Parameters

Name	Type	Description
systemCode	String	The system code of the system object to deactivate.
localid	String	The local ID of the system object to deactivate.

Returns

None.

Throws

- ProcessingException
- UserException

executeMatch

executeMatch is one of two methods you can call to process an incoming system object based on the configuration defined for the Manager Service and associated runtime components (the second method is [“executeMatchUpdate” on page 71](#)). This process searches for possible matches in the database and contains the logic to add a new record or update existing records in the database. One of the two execute match methods should be used for inserting or updating a record in the database.

The following runtime components configure executeMatch.

- The Query Builder defines the blocking queries used for matching.
- The Threshold file (master.xml) specifies which blocking query to use and specifies matching parameters, including duplicate and match thresholds.
- The pass controller and block picker classes specify how the blocking query is executed.

Note – If `executeMatch` determines that an existing system record will be updated by the incoming record, it replaces the entire existing record with the information in the new record. This could result in loss of data; for example, if the incoming record does not include all address information, existing address information could be lost. To avoid this, use the `executeMatchUpdate` method instead.

Syntax

```
MatchColResult executeMatch(SystemObjectName sysObjBean)
```

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
sysObjBean	<i>SystemObjectName</i>	The Bean for the system object to be added to or updated in the enterprise object.

Returns

A match result object containing the results of the matching process.

Throws

- `ProcessingException`
- `UserException`

executeMatchUpdate

Like “[executeMatch](#)” on page 70, `executeMatchUpdate` processes the system object based on the configuration defined for the Manager Service and associated runtime components. It is configured by the same runtime components as `executeMatch`. One of these two execute match methods should be used for inserting or updating a record in the database.

The primary difference between these two methods is that when `executeMatchUpdate` finds that an incoming record matches an existing record, only the changed data is updated. With `executeMatch`, the entire existing record would be replaced by the incoming record. The `executeMatchUpdate` method differs from `executeMatch` in the following ways:

- If a partial record is received, `executeMatchUpdate` only updates fields whose values are different in the incoming record. Unless the `clearFieldIndicator` field is used, empty or null fields in the incoming record do not update existing values.
- The `clearFieldIndicator` field can be used to null out specific fields.

- Child objects in the existing record are not deleted if they are not present in the incoming record.
- Child objects in the existing record are updated if the same key field value is found in both the incoming and existing records.
- To allow a child object to be removed from the parent object when using `executeMatchUpdate`, a new “delete” method is added to each child object bean.

Syntax

`MatchColResult executeMatchUpdate(SystemObjectName sysObjBean)`

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
<code>sysObjBean</code>	<i>SystemObjectName</i>	The Bean for the system object to be added to or updated in the enterprise object.

Returns

A match result object containing the results of the matching process.

Throws

- `ProcessingException`
- `UserException`

getEnterpriseRecordByEUID

Description

This method returns the enterprise object associated with the specified EUID.

Syntax

`EnterpriseObjectName getEnterpriseRecordByEUID(String euid)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object you want to retrieve.

Returns

An enterprise object associated with the specified EUID or null if the enterprise object is not found.

Throws

- `ProcessingException`
- `UserException`

getEnterpriseRecordByLID

Description

This method returns the enterprise object associated with the specified system code and local ID pair.

Syntax

```
EnterpriseObjectName getEnterpriseRecordByLID(String systemCode, String localid)
```

Parameters

Name	Type	Description
systemCode	String	The system code of a system associated with the enterprise object to find.
localid	String	A local ID associated with the specified system.

Returns

An enterprise object or null if the enterprise object is not found.

Throws

- `ProcessingException`
- `UserException`

getEUID

Description

This method returns the EUID of the enterprise object associated with the specified system code and local ID.

Syntax

String getEUID(String systemCode, String localid)

Parameters

Name	Type	Description
systemCode	String	A known system code for the enterprise object.
localid	String	The local ID corresponding with the given system.

Returns

A string containing an EUID or null if the EUID is not found.

Throws

- ProcessingException
- UserException

getLIDs

Description

This method retrieves the local ID and system pairs associated with the given EUID.

Syntax

SystemObjectNamePK[] getLIDs(String euid)

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose local ID and system pairs you want to retrieve.

Returns

An array of system object keys (*SystemObjectNamePK* objects) or null if no results are found.

Throws

- *ProcessingException*
- *UserException*

getLIDsByStatus

Description

This method retrieves the local ID and system pairs that are of the specified status and that are associated with the given EUID.

Syntax

```
SystemObjectNamePK[] getLIDsByStatus(String euid, String status)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose local ID and system pairs to retrieve.
status	String	The status of the local ID and system pairs to retrieve.

Returns

An array of system object keys (*SystemObjectNamePK* objects) or null if no system object keys are found.

Throws

- *ProcessingException*
- *UserException*

getSBR

Description

This method retrieves the single best record (SBR) associated with the specified EUID.

Syntax

SBRObjectName getSBR(String *euid*)

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose SBR you want to retrieve.

Returns

An SBR object or null if no SBR associated with the specified EUID is found.

Throws

- `ProcessingException`
- `UserException`

getSystemRecord

Description

This method retrieves the system object associated with the given system code and local ID pair.

Syntax

SystemObjectName getSystemRecord(String *systemCode*, String *localid*)

Parameters

Name	Type	Description
systemCode	String	The system code of the system object to retrieve.
localid	String	The local ID of the system object to retrieve.

Returns

A system object containing the results of the search or null if no system objects are found.

Throws

- `ProcessingException`
- `UserException`

getSystemRecordsByEUID

Description

This method returns the active system objects associated with the specified EUID.

Syntax

```
SystemObjectName[] getSystemRecordsByEUID(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose system objects you want to retrieve.

Returns

An array of system objects associated with the specified EUID.

Throws

- ProcessingException
- UserException

getSystemRecordsByEUIDStatus

Description

This method returns the system objects of the specified status that are associated with the given EUID.

Syntax

```
SystemObjectName[] getSystemRecordsByEUIDStatus(String euid, String status)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose system objects you want to retrieve.

Name	Type	Description
status	String	The status of the system objects you want to retrieve.

Returns

An array of system objects associated with the specified EUID and status, or null if no system objects are found.

Throws

- `ProcessingException`
- `UserException`

lookupLIDs

Description

This method first looks up the EUID associated with the specified source system and source local ID. It then retrieves the local ID and system pairs of the specified status that are associated with that EUID and are from the specified destination system. Note that both systems must be of the specified status or an error will occur.

Syntax

```
SystemObjectNamePK[] lookupLIDs(String sourceSystemCode, String sourceLID,  
String destSystemCode, String status)
```

Parameters

Name	Type	Description
sourceSystemCode	String	The system code of the known system and local ID pair.
sourceLID	String	The local ID of the known system and local ID pair.
destSystemCode	String	The system from which the local ID and system pairs to retrieve originated.
status	String	The status of the local ID and system pairs to retrieve.

Returns

An array of system object keys (`SystemObjectNamePK` objects).

Throws

- `ProcessingException`
- `UserException`

lookupPotentialDuplicates

Description

This method returns an array of potential duplicate result records based on the criteria contained in the potential duplicate search object.

Syntax

```
PotentialDuplicateResult[] lookupPotentialDuplicates(PotentialDuplicateSearchObjectBean pdsoBean)
```

Parameters

Name	Type	Description
pdsoBean	PotentialDuplicateSearchObjectBean	An object containing the criteria for a potential duplicate search.

Returns

An array of `PotentialDuplicateResult` objects representing the matches to a potential duplicate search.

Throws

- `ProcessingException`
- `UserException`
- `PageException`
- `RemoteException`

mergeEnterpriseRecord

Description

This method merges two enterprise objects, specified by their EUIDs.

Syntax

```
MergeObjectNameResult mergeEnterpriseRecord(String fromEUID, String toEUID,  
boolean calculateOnly)
```

Parameters

Name	Type	Description
fromEUID	String	The EUID of the enterprise object that will not survive the merge.
toEUID	String	The EUID of the enterprise object that will survive the merge.
calculateOnly	boolean	An indicator of whether to commit changes to the database or to simply compute the merge results. Specify false to commit the changes.

Returns

A merge result object containing the results of the merge.

Throws

- `ProcessingException`
- `UserException`

mergeSystemRecord

Description

This method merges two system objects, specified by their local IDs, from the specified system. The system objects can belong to a single enterprise object or to two different enterprise objects.

Syntax

```
MergeObjectNameResult mergeSystemRecord(String systemCode, String sourceLID,  
String destLID, boolean calculateOnly)
```

Parameters

Name	Type	Description
systemCode	String	The processing code of the system to which the two system objects belong.
sourceLID	String	The local ID of the system object that will not survive the merge.
destLID	String	The local ID of the system object that will survive the merge.

Name	Type	Description
calculateOnly	boolean	An indicator of whether to commit changes to the database or to simply compute the merge results. Specify false to commit the changes.

Returns

A merge result object containing the results of the merge.

Throws

- `ProcessingException`
- `UserException`

searchBlock

Description

This method performs a blocking query against the database using the blocking query specified in the `master.xml` and the criteria contained in the specified object bean.

Syntax

```
SearchObjectNameResult[] searchBlock(ObjectNameBean objBean)
```

Parameters

Name	Type	Description
objBean	<i>ObjectNameBean</i>	The search criteria for the blocking query.

Returns

An array of search result records.

Throws

- `ProcessingException`
- `UserException`

searchExact

Description

This method performs an exact match search using the criteria specified in the object bean. Only records that exactly match the search criteria are returned in the search results object.

Syntax

```
SearchObjectNameResult[] searchExact(ObjectNameBean objBean)
```

Parameters

Name	Type	Description
objBean	<i>ObjectNameBean</i>	The search criteria for the exact match search.

Returns

An array of search result records.

Throws

- *ProcessingException*
- *UserException*

searchPhonetic

Description

This method performs search using phonetic values for some of the criteria specified in the object bean. This type of search allows for typographical errors and misspellings.

Syntax

```
SearchObjectNameResult[] searchPhonetic(ObjectNameBean objBean)
```

Parameters

Name	Type	Description
objBean	<i>ObjectNameBean</i>	The search criteria for the phonetic search.

Returns

An array of search result records.

Throws

- `ProcessingException`
- `UserException`

transferSystemRecord

Description

This method transfers a system record from one enterprise record to another enterprise record.

Syntax

```
void transferSystemRecord(String toEUID, String systemCode, String localID)
```

Parameters

Name	Type	Description
toEUID	String	The EUID of the enterprise record to which the system record will be transferred.
systemCode	String	The processing code of the system record to transfer.
localID	String	The local ID of the system record to transfer.

Returns

None.

Throws

- `ProcessingException`
- `UserException`

updateEnterpriseRecord

Description

This method updates the fields in an existing enterprise object with the values specified in the fields the enterprise object passed in as a parameter. When updating an enterprise object, attempting to change a field that is not updateable will cause an exception. This method does not update the SBR; the survivor calculator updates the SBR once the changes are made to the associated system records.

Syntax

```
void updateEnterpriseRecord(EnterpriseObjectName eoBean)
```

Parameters

Name	Type	Description
eoBean	EnterpriseObjectName	The enterprise object containing the values that will update the existing enterprise object.

Returns

None.

Throws

- ProcessingException
- UserException

updateSystemRecord

Description

This method updates the existing system object in the database with the given system object.

Syntax

```
void updateSystemRecord(SystemObjectName sysObjBean)
```

Parameters

Name	Type	Description
sysObjBean	<i>SystemObjectName</i>	The system object to be updated to the enterprise object.

Returns

None.

Throws

- `ProcessingException`
- `UserException`

Web Services Helper Classes

Helper classes include objects that can be passed as parameters to a primary web services method. They also include the methods that you can access through the `systemObjectName` variable in Java clients or Business Processes (where *ObjectName* is the name of a parent object). The helper classes include:

- “*SystemObjectName*” on page 95
- “Parent Beans” on page 100
- “Child Beans” on page 108
- “DestinationEO” on page 113
- “*SearchObjectNameResult*” on page 121
- “SourceEO” on page 124
- “*SystemObjectNamePK*” on page 126

Helper classes are located in the Master Index EJB project in the `com.sun.mdm.index.webservice` package.

Enterprise Object Classes

A Java class is created to represent the entire object, which includes the single best record (SBR) and any associated system objects. The methods in this class provide the ability to retrieve, set, and delete system objects, and to retrieve and set attributes of the enterprise object.

The name of the enterprise object class is the same as the name of the application, with the word “Enterprise” prepended. For example, if the application name is “Person”, the name of the enterprise object class is “EnterprisePerson”. The methods in this class include a constructor

method for the enterprise object, get and set methods for object attributes, and get, set, and delete methods for the system objects. Many methods have dynamic names based on the name of the parent object. In the following methods described for the enterprise object, *ObjectName* indicates the name of the parent object.

Definition

```
class ObjectNameObject
```

Methods

- “*EnterpriseObjectName*” on page 86
- “*countSystemObjectName*” on page 87
- “*deleteSystemObject*” on page 87
- “*deleteSystemObjectBean*” on page 88
- “*getEUID*” on page 89
- “*getSBRObjName*” on page 89
- “*getStatus*” on page 90
- “*getSystemObjectName*” on page 90
- “*getSystemObjectName*” on page 91
- “*getSystemObjectNameCount*” on page 91
- “*removeSystemObject*” on page 92
- “*removeSystemObjectBean*” on page 92
- “*setEUID*” on page 93
- “*setStatus*” on page 94
- “*setSystemObjectName*” on page 94

EnterpriseObjectName

Description

This is the constructor method for the enterprise object class. You can instantiate this class to create a new instance of the enterprise object.

Syntax

```
new EnterpriseObjectName()
```

Parameters

None.

Returns

An instance of the enterprise object.

Throws

None.

countSystemObjectName

Description

This method counts the system objects (class *SystemObjectName*) in the enterprise object. Use this method to loop through the system objects.

Syntax

```
int countSystemObjectName()
```

Parameters

None.

Returns

An integer representing the number of system objects.

Throws

None.

deleteSystemObject

Description

This method marks a system object for deletion from an instance of *EnterpriseObjectName*. The system object is deleted when the enterprise object is updated.

Syntax

```
void deleteSystemObject(String system, String lid)
```

Parameters

Name	Type	Description
system	String	The processing code of the system associated with the system record to delete.

Name	Type	Description
lid	String	The local ID of the system record in the specified system.

Returns

None.

Throws

ObjectException

deleteSystemObjectBean

Description

This method marks a system object for deletion from an instance of *EnterpriseObjectName*. The system object is deleted when the enterprise object is updated.

Syntax

```
void deleteSystemObjectBean(String system, String lid)
```

Parameters

Name	Type	Description
system	String	The processing code of the system associated with the system record to delete.
lid	String	The local ID of the system record in the specified system.

Returns

None.

Throws

ObjectException

getEUID

Description

This method retrieves the EUID from an instance of *EnterpriseObjectName*.

Syntax

```
String getEUID()
```

Parameters

None.

Returns

A string containing the EUID of the enterprise object.

Throws

ObjectException

getSBRObjObjectName

Description

This method retrieves the SBR from an instance of *EnterpriseObjectName*.

Syntax

```
SBRObjectName getSBRObjObjectName()
```

Parameters

None.

Returns

The SBR object from the enterprise object.

Throws

None.

getStatus

Description

This method retrieves the status of the enterprise object.

Syntax

```
String getStatus()
```

Parameters

None.

Returns

The status of the enterprise object.

Throws

ObjectException

getSystemObjectName

Description

This method retrieves all of the system objects associated with the enterprise object.

Syntax

```
SystemObjectName[] getSystemObjectName()
```

Parameters

None.

Returns

A collection of system objects.

Throws

None.

getSystem*ObjectName*

Description

This method retrieves a system object based on its index in a list of system objects.

Syntax

```
SystemObjectName getSystemObjectName(int index)
```

Parameters

Name	Type	Description
index	int	The index of the system object to retrieve from a list of system objects.

Returns

The system object at the specified index.

Throws

None.

getSystem*ObjectNameCount*

Description

This method returns the number of system objects in a collection of system objects.

Syntax

```
int getSystemObjectNameCount()
```

Parameters

None.

Returns

An integer indicating the number of system objects in an enterprise object.

Throws

None.

removeSystemObject

Description

This method removes a system object from an enterprise object. Use this to remove the system object from the in-memory instance of *EnterpriseObjectName*. This does NOT result in the deletion of the system object from the database (see *deleteSystemObject*).

Syntax

```
void removeSystemObject(String system, String lid)
```

Parameters

Name	Type	Description
system	String	The processing code of the system associated with the system object to remove.
lid	String	The local ID of the system record in the specified system.

Returns

None.

Throws

ObjectException

removeSystemObjectBean

Description

This method removes a system object from an enterprise object. Use this to remove the system object from the in-memory instance of *EnterpriseObjectName*. This does NOT result in the deletion of the system object from the database (see *deleteSystemObjectBean*).

Syntax

```
void removeSystemObjectBean(String system, String lid)
```

Parameters

Name	Type	Description
system	String	The processing code of the system associated with the system object to remove.
lid	String	The local ID of the system record in the specified system.

Returns

None.

Throws

ObjectException

setEUID

Description

This method sets the value of the EUID field in an enterprise object.

Syntax

```
void setEUID(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object.

Returns

None.

Throws

ObjectException

setStatus

Description

This method sets the status of the enterprise object.

Syntax

```
void setStatus(String status)
```

Parameters

Name	Type	Description
status	String	The status of the enterprise object.

Returns

None.

Throws

ObjectException.

setSystemObjectName

Description

This method sets a collection of system objects in an enterprise object.

Syntax

```
void setSystemObjetName(SystemObjectName[] systemObjectName)
```

Parameters

Name	Type	Description
<i>systemObjectName</i>	<i>SystemObjectName</i>	A collection of system objects for the enterprise object.

Returns

None.

Throws

ObjectException.

SystemObjectName

In order to run `executeMatch` or `executeMatchUpdate` from a client application, you must define a variable of the class type `SystemObjectName`, where *ObjectName* is the name of a parent object. This class is passed as a parameter to the execute match methods. The class contains a constructor method and several get and set methods for system fields. It also includes one field that specifies the value of the clear field character (for more information, see [“Clear Field Indicator” on page 96](#)).

In the methods described in this section, *ObjectName* indicates the name of the parent object, *Child* indicates the name of a child object, and *Field* indicates the name of a field defined for the parent object.

Definition

```
class SystemObjectName
```

Methods

- [“SystemObjectName” on page 96](#)
- [“getClearFieldIndicator” on page 96](#)
- [“getField” on page 97](#)
- [“getObjectName” on page 97](#)
- [“setClearFieldIndicator” on page 98](#)
- [“setField” on page 99](#)
- [“setObjectName” on page 99](#)

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- `equals`
- `hashCode`
- `notify`
- `notifyAll`
- `toString`
- `wait()`
- `wait(long arg)`
- `wait(long timeout, int nanos)`

Clear Field Indicator

The clear field indicator allows you to specify whether to treat a field in the parent object as null when performing an update from an external system. When an update is performed in the master index application, empty fields typically do not overwrite the value of an existing field. You can specify to nullify a field that already has an existing value in the master index application by entering an indicator in that field. This indicator is specified by the clear field indicator. By default, the clear field indicator is set to double-quotes (“”), so if a field is set to double-quotes, that field will be blanked out in the master index if the incoming record has a field value of null and the record being updated has a field value other than null. If you do not want to use this feature, set the clear field indicator to null.

System*ObjectName*

Description

This method is the user-defined system class for the parent object. You can instantiate this class to create a new instance of the system class.

Syntax

```
new SystemObjectName()
```

Parameters

None.

Returns

An instance of the *SystemObjectName* class.

Throws

ObjectException

getClearFieldIndicator

Description

This method retrieves the value of the clear field indicator.

Syntax

```
Object getClearFieldIndicator()
```


Parameters

None.

Returns

The clear field indicator.

Throws

None.

getField

Description

This method retrieves the value of the specified system field. There are getter methods for the following fields: LocalId, SystemCode, Status, CreateDateTime, CreateFunction, CreateUser, and UpdateUser.

Syntax

```
String getField()
```

or

```
Date getField()
```

Parameters

None.

Returns

The value of the specified field. The type of value returned depends on the field from which the value was retrieved.

Throws

ObjectException

getObjectName

Description

This method retrieves the system record JavaBean.

Syntax

```
ObjectNameBean getObjectName()
```

Parameters

None.

Returns

A JavaBean containing the system record.

Throws

None.

setClearFieldIndicator

Description

This method sets the value of the clear field indicator. By default, this is set to double quotes (“”).

Syntax

```
void setClearFieldIndicator(String value)
```

Parameters

Name	Type	Description
value	String	The value that should be entered into a field to indicate that any existing values should be replaced with null.

Returns

None.

Throws

None.

setField

Description

This method sets the value of the specified system field. There are setter methods for the following fields: LocalId, SystemCode, Status, CreateDateTime, CreateFunction, CreateUser, and UpdateUser.

Syntax

```
void setField(value)
```

Parameters

Name	Type	Description
value	varies	The value to set in the specified field. The type of value depends on the field into which the value is being set.

Returns

None.

Throws

ObjectException

setObjectName

Description

This method sets the system object JavaBean.

Syntax

```
void setObjectName(ObjectNameBean aObjectName)
```

Parameters

Name	Type	Description
aObjectName	ObjectNameBean	The JavaBean for the parent object.

Returns

None.

Throws

ObjectException

Parent Beans

A JavaBean is created to represent the parent object defined in the object structure of the master index application. The methods in this class are generated based on the object structure and provide the ability to create a parent object Bean and to set or retrieve the field values for that JavaBean.

The name of the parent JavaBean class is the same as the name of the parent object, with the word “Bean” appended. For example, if the parent object in your object structure is “Person”, the name of the associated parent Bean class is “PersonBean”. The methods in this class include a constructor method for the parent object Bean, get and set methods for each field defined for the parent object, and methods to manipulate child objects. Most methods have dynamic names based on the name of the parent object and the fields and child objects defined for that object. In the methods described in this section, *ObjectName* indicates the name of the parent object, *Child* indicates the name of a child object, and *Field* indicates the name of a field defined for the parent object.

Definition

```
final class ObjectNameBean
```

Methods

- “*ObjectNameBean*” on page 101
- “*countChild*” on page 101
- “*countChildren*” on page 102
- “*countChildren*” on page 102
- “*deleteChild*” on page 103
- “*getChild*” on page 104
- “*getChild*” on page 104
- “*getField*” on page 105
- “*getObjectNameId*” on page 105
- “*setChild*” on page 106
- “*setChild*” on page 106
- “*setField*” on page 107
- “*setObjectNameId*” on page 108

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- `equals`
- `hashCode`
- `notify`
- `notifyAll`
- `toString`
- `wait()`
- `wait(long arg)`
- `wait(long timeout, int nanos)`

*ObjectName***Bean**

Description

This method is the user-defined object Bean class. You can instantiate this class to create a new instance of the parent object Bean class.

Syntax

```
new ObjectNameBean()
```

Parameters

None.

Returns

An instance of the parent object Bean.

Throws

`ObjectException`

count*Child*

Description

This method returns the total number of child objects contained in a system object. The type of child object is specified by the method name, such as `countPhone` or `countAddress`.

Syntax

```
int countChild()
```

Parameters

None.

Returns

An integer indicating the number of child objects in a collection.

Throws

None.

countChildren

Description

This method returns a count of the total number of child objects belonging to a system object.

Syntax

```
int countChildren()
```

Parameters

None.

Returns

An integer representing the total number of child objects.

Throws

None.

countChildren

Description

This method returns a count of the total number of child objects of a specific type that belong to a system object.

Syntax

```
int countChildren(String type)
```

Parameters

Name	Type	Description
type	String	The type of child object to count, such as Phone or Address.

Returns

An integer representing the total number of child objects of the specified type.

Throws

None.

delete*Child*

Description

This method removes the specified child object from the system object. The type of child object to remove is specified by the name of the method, and the specific child object to remove is specified by its unique identification code assigned by the master index application.

Syntax

```
void deleteChild(String ChildId)
```

Parameters

Name	Type	Description
<i>ChildId</i>	String	The unique identification code of the child object to delete.

Returns

None.

Throws

ObjectException

getChild

Description

This method retrieves an array of child object JavaBeans. Each getter method is named according to the child objects defined for the parent object. For example, if the parent object contains a child object named “Address”, the getter method for this object is named getAddress. A getter method is created for each child object in the parent object.

Syntax

ChildBean[] get*Child*()

Parameters

None.

Returns

An array of JavaBeans containing the type of child object specified by the method name.

Throws

None.

getChild

Description

This method retrieves a child object JavaBean based on its index in a list of child objects. Each getter method is named according to the child objects defined for the parent object. For example, if the parent object contains a child object named “Address”, the getter method for this field is named getAddress. A getter method is created for each child object in the parent object.

Syntax

ChildBean get*Child*(int i)

Parameters

Name	Type	Description
i	int	The index of the child object to retrieve from a list of child objects.

Returns

A `JavaBean` containing the child object specified by the index value. The method name indicates the type of child object returned.

Throws

`ObjectException`

getField

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named “`FirstName`”, the getter method for this field is named `getFirstName`.

Syntax

```
String getField()
```

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

```
Date getField()
```

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object structure.

Throws

`ObjectException`

getObjectNamed

Description

This method retrieves the unique identification code (primary key) of the parent object, as assigned by the master index application.

Syntax

```
String getObjectNamedId()
```

Parameters

None.

Returns

A string containing the unique ID of the parent object.

Throws

ObjectException

setChild

Description

This method adds a child object to the system object.

Syntax

```
void setChild(int index, ChildBean child)
```

Parameters

Name	Type	Description
index	integer	The index number for the new child object.
child	ChildBean	The JavaBean containing the child object to add.

Returns

None.

Throws

None.

setChild

Description

This method adds an array of child objects of one type to the system object.

Syntax

```
void setChild(ChildBean[] children)
```

Parameters

Name	Type	Description
children	ChildBean[]	The array of child objects to add.

Returns

None.

Throws

None.

set*Field*

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named “DateOfBirth”, the setter method for this field is named `setDateOfBirth`. A setter method is created for each field in the parent object, including any fields containing standardized or phonetic data.

Syntax

```
void setField(value)
```

Parameters

Name	Type	Description
value	varies	The value of the field specified by the method name. The type of value depends on the field being populated.

Returns

None.

Throws

ObjectException

setObjectNameId

Description

This method sets the value of the *ObjectNameId* field in the parent object.

Note – This ID is set internally by the master index application. You should never set this field manually.

Syntax

```
void setObjectNameId(String value)
```

Parameters

Name	Type	Description
value	String	The value of the <i>ObjectNameId</i> field.

Returns

None.

Throws

ObjectException

Child Beans

A *JavaBean* is created to represent each child object defined in the object structure of the master index application. The methods in these classes provide the ability to create a child object *Bean* and to set or retrieve the field values for that object *Bean*.

The name of each child object *JavaBean* class is the same as the name of each child object, with the word “*Bean*” appended. For example, if a child object in your object structure is named “*Address*”, the name of the corresponding child class is *AddressBean*. The methods in this class include a constructor method for the child object *JavaBean*, and get and set methods for each field defined for the child object. Most methods have dynamic names based on the name of the child object and the fields defined for that object. In the following methods, *Child* indicates the name of a child object and *Field* indicates the name of a field defined for the child object.

Definition

`final class ChildBean`

Methods

- “*ChildBean*” on page 109
- “*delete*” on page 110
- “*getField*” on page 110
- “*getChildId*” on page 111
- “*setField*” on page 112
- “*setChildId*” on page 112

Note – These JavaBeans also include two `countChildren` methods, which are not currently used.

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- `equals`
- `hashCode`
- `notify`
- `notifyAll`
- `toString`
- `wait()`
- `wait(long arg)`
- `wait(long timeout, int nanos)`

ChildBean

Description

This method is the constructor for the user-defined object JavaBean class. You can instantiate this class to create a new instance of the JavaBean.

Syntax

`new ChildBean()`

Parameters

None.

Returns

An instance of the child object Bean.

Throws

ObjectException

delete

Description

This method removes the child object from its corresponding parent object. This is used with the `executeMatchUpdate` function to update a system object by deleting one of the child objects from the object being processed.

Syntax

```
void delete()
```

Parameters

None.

Returns

None.

Throws

ObjectException

get*Field*

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the child object. For example, if the child object contains a field named “ZipCode”, the getter method for this field is named `getZipCode`.

Syntax

```
String getField()
```

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

Date *getField*

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object structure.

Throws

ObjectException

getChildId

Description

This method retrieves the unique identification code (primary key) of the object, as assigned by the master index application.

Syntax

String *getChildId()*

Parameters

None.

Returns

A string containing the unique ID of the child object.

Throws

ObjectException

setField

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the child object. For example, if the child object contains a field named “Address”, the setter method for this field is named setAddress. A setter method is created for each field in the child object, including any fields containing standardized or phonetic data.

Syntax

```
void setField(value)
```

Parameters

Name	Type	Description
value	varies	The value of the field specified by the method name. The type of value depends on the data type of the field being populated.

Returns

None.

Throws

ObjectException

setChildId

Description

This method sets the value of the *ChildId* field in the child object.

Note – This ID is set internally by the master index application. You should never set this field manually.

Syntax

```
void setChildId(String value)
```


Parameters

Name	Type	Description
value	String	The value of the <i>ChildId</i> field.

Returns

None.

Throws

ObjectException

DestinationEO

This class represents an enterprise object involved in a merge. This is the enterprise object whose EUID was kept in the final merge result record. A DestinationEO object is used when unmerging two enterprise objects.

Definition

```
class DestinationEO
```

Methods

- [“getEnterpriseObjectName” on page 113](#)
- [“setEnterpriseObjectName” on page 114](#)

getEnterpriseObjectName

Description

This method retrieves the surviving enterprise object from a merge transaction in order to allow the records to be unmerged.

Syntax

```
EnterpriseObjectName getEnterpriseObjectName()
```

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

The surviving enterprise object from a merge transaction.

Throws

ObjectException

setEnterpriseObjectName

Description

This method sets the surviving enterprise object of a merge transaction.

Syntax

`void setEnterpriseObjectName(EnterpriseObjectName eo)`

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
eo	EnterpriseObjectName	The destination enterprise object to set.

Returns

None.

Throws

ObjectException

MergeObjectNameResult

This class represents the return values for a call to the merge-related functions of the *ObjectNameEJB* class, where *ObjectName* is the name of the parent object.

Definition

`class MergeObjectNameResult`

Methods

- [“MergeObjectNameResult” on page 115](#)
- [“getDestinationEO” on page 115](#)
- [“getSourceEO” on page 116](#)
- [“setDestinationEO” on page 116](#)
- [“setSourceEO” on page 117](#)

MergeObjectNameResult

Description

This is the constructor method for the *MergeObjectNameResult* class.

Syntax

```
new MergeObjectnameResult()
```

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

A new instance of *MergeObjectNameResult*.

Throws

None.

getDestinationEO

Description

This method retrieves the surviving enterprise object from a merge result.

Syntax

```
DestinationEO getDestinationEO()
```

Parameters

None.

Returns

The surviving enterprise object from a merge result.

Throws

None.

getSourceEO

Description

This method retrieves the nonsurviving enterprise object from a merge result.

Syntax

```
SourceEO getSourceEO()
```

Parameters

None.

Returns

The nonsurviving enterprise object from a merge result.

Throws

None.

setDestinationEO

Description

This method sets the surviving enterprise object in a merge result.

Syntax

```
void setDestinationEO(DestinationEO eom)
```

Parameters

Name	Type	Description
eom	DestinationEO	The surviving record in the merge result.

Returns

None.

Throws

None.

setSourceEO

Description

This method sets the nonsurviving enterprise object in a merge result.

Syntax

```
void setSourceEO(SourceEO eom)
```

Parameters

Name	Type	Description
eom	SourceEO	The nonsurviving record in the merge result.

Returns

None.

Throws

None.

SBR*ObjectName*

This class represents an SBR object in an enterprise object. The SBR (single best record) represents the data about an entity that is considered to be the most accurate information from all external systems. Many of the methods in this class are named based on the name of the parent object, which is represented by *ObjectName* in the method descriptions below.

Definition

`class SBRObjectName`

Methods

- [“SBRObjectName” on page 118](#)
- [“SBRObjectName” on page 118](#)
- [“getObjectName” on page 119](#)
- [“getStatus” on page 119](#)
- [“setObjectName” on page 120](#)
- [“setStatus” on page 120](#)

SBRObjectName

Description

This is a constructor method for an empty *SBRObjectName* class.

Syntax

`new SBRObjectName()`

Parameters

None.

Returns

A new instance of *SBRObjectName*.

Throws

None.

SBRObjectName

Description

This is a constructor method for an *SBRObjectName* class. It takes an SBR object as its input.

Syntax

`new SBRObjectName(SBR sbr)`

Parameters

Name	Type	Description
sbr	<i>SBRObjectName</i>	The SBR object to use to construct the new instance of <i>SBRObjectName</i> .

Returns

None.

Throws

ObjectException

getObjectName

Description

This method retrieves the parent object in the SBR.

Syntax

```
ObjectNameBean getObjectName()
```

Parameters

None.

Returns

The *ObjectNameBean* object associated with the SBR.

Throws

ObjectException

getStatus

Description

This method retrieves the status of the SBR.

Syntax

```
String getStatus()
```

Parameters

None.

Returns

The status of the SBR.

Throws

ObjectException

setObjectName

Description

This method sets the parent object in the SBR.

Syntax

`void setObjectName(ObjectNameBean aObjectName)`

Parameters

Name	Type	Description
<i>aObjectName</i>	<i>ObjectNameBean</i>	The parent object bean to set in the SBR.

Returns

None.

Throws

ObjectException

setStatus

Description

This method sets the status of the SBR.

Syntax

`void setStatus(String status)`

Parameters

Name	Type	Description
status	String	The status of the SBR.

Returns

None.

Throws

ObjectException

SearchObjectNameResult

This class represents the results of a search. A *SearchObjectNameResult* object (where *ObjectName* is the name of the parent object) is returned as a result of a call to [“searchBlock” on page 81](#), [“searchExact” on page 82](#), or [“searchPhonetic” on page 82](#). [“searchBlock” on page 81](#)

Definition

```
class SearchObjectNameResult
```

Methods

- [“getEUID” on page 121](#)
- [“getComparisonScore” on page 122](#)
- [“getObjectName” on page 122](#)
- [“setEUID” on page 123](#)
- [“setComparisonScore” on page 123](#)
- [“setObjectName” on page 124](#)

getEUID

Description

This method retrieves the EUID of a search result record.

Syntax

```
String getEUID()
```

Parameters

None.

Returns

A string containing an EUID.

Throws

None.

getComparisonScore

Description

This method retrieves the weight that indicates how closely a search result record matched the search criteria.

Syntax

```
Float getComparisonScore()
```

Parameters

None.

Returns

A comparison weight.

Throws

None.

getObjectName

Description

This method retrieves an object bean for a search result record.

Syntax

```
ObjectNameBean getObjectName()
```

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

An object bean.

Throws

None.

setEUID

Description

This method sets the EUID of a search result record.

Syntax

```
void setEUID(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID to set.

Returns

None.

Throws

None.

setComparisonScore

Description

This method sets the weight that indicates how closely a search result record matched the search criteria.

Syntax

```
void setComparisonScore(float score)
```

Parameters

Name	Type	Description
score	Float	The matching comparison score to set.

Returns

None.

Throws

None.

set*ObjectName*

Description

This method sets an object bean as a search result record.

Syntax

```
void setObjectName(ObjectNameBean bean)
```

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
bean	<i>ObjectNameBean</i>	The object bean to set.

Returns

None.

Throws

None.

SourceEO

This class represents an enterprise object involved in a merge. This is the enterprise object whose EUID was not kept in the final merge result record. A SourceEO object is used when unmerging two enterprise objects.

Definition

class SourceEO

Methods

- [“getEnterpriseObjectName” on page 125](#)
- [“SetEnterpriseObjectName” on page 125](#)

getEnterpriseObjectName

Description

This method retrieves the nonsurviving enterprise object from a merge transaction in order to allow the records to be unmerged.

Syntax

`EnterpriseObjectName getEnterpriseObjectName()`

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

The nonsurviving enterprise object from a merge transaction.

Throws

None.

SetEnterpriseObjectName

Description

This method sets the nonsurviving enterprise object from a merge transaction in order to allow the records to be unmerged.

Syntax

`void setEnterpriseObjectName(EnterpriseObjectName eo)`

where *ObjectName* is the name of the parent object.

Parameters

Name	Type	Description
eo	EnterpriseObjectName	The enterprise object to set.

Returns

None.

Throws

ObjectException.

SystemObjectNamePK

This class represents the primary keys in a system object, which include the processing code for the originating system and the local ID of the object in that system. The class is named for the primary object. For example, if the primary object is named “Person”, this class is named SystemPersonPK. If the primary object is named “Company”, this class is named SystemCompanyPK. The methods in these classes provide the ability to create an instance of the class and to retrieve the system processing code and the local ID.

Definition

```
class SystemObjectNamePK
```

where *ObjectName* is the name of the parent object.

Methods

- [“SystemObjectNamePK” on page 126](#)
- [“getLocalId” on page 127](#)
- [“getSystemCode” on page 127](#)

SystemObjectNamePK

Description

This method is the user-defined system primary key object. This object contains a system code and a local ID. Use this constructor method to create a new instance of a system primary key object.

Syntax

```
new SystemObjectNamePK()
```

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

An instance of the system primary key object.

Throws

None.

getLocalId

Description

This method retrieves the local identifier from a system primary key object.

Syntax

```
String getLocalId()
```

Parameters

None.

Returns

A string containing a local identifier.

Throws

None.

getSystemCode

Description

This method retrieves the system's processing code from a system primary key object.

Syntax

```
String getSystemCode()
```

Parameters

None.

Returns

A string containing the processing code for a system.

Throws

None.

Inbound Message Processing with Custom Logic

You can customize the way the execute match methods process inbound messages by defining custom plug-ins that include decision-point methods. This appendix describes the standard inbound processing logic as described in [“Inbound Message Processing Logic” on page 15](#), but also includes how the decision-point methods alter the process.

Custom Decision Point Logic

There are several decision points in the match process that can be defined by custom logic using custom plug-ins. The steps below are identical to those outlined in [“Inbound Message Processing Logic” on page 15](#), but include descriptions of the decision points, which are listed in italic font. If no custom logic is defined, the decisions default to “false”, and processing is identical to that described in [“Inbound Message Processing Logic” on page 15](#).

For more information about the methods and plug-ins, see [“Master Index Match Processing Logic Plug-ins” in *Sun Master Index User’s Guide*](#). For detailed information about the methods, see the Javadocs provided with Sun Master Index. The methods are contained in the `ExecuteMatchLogics` class in the package `com.sun.mdm.index.master`.

1. When a message is received by the master index application, a search is performed for any existing records with the same local ID and system as those contained in the message. This search only includes records with a status of *A*, meaning only active records are included. If a matching record is found, an existing EUID is returned.
2. If an existing record is found with the same system and local ID as the incoming message, it is assumed that the two records represent the same entity. Using the EUID of the existing record, the master index application performs an update of the record’s information in the database.

Custom plug-in decision point: If `disallowUpdate` is set to true, the update is not allowed and a `MatchResult` object is returned with a result code of 12. If `disallowUpdate` is set to false and `rejectUpdate` is set to true, the update is not allowed and a `MatchResult` object is returned with a result code of 13.

- If the update does not make any changes to the object's information, no further processing is required and the existing EUID is returned.
 - If there are changes to the object's information, the updated record is inserted into database, and the changes are recorded in the `sbyn_transaction` table.
 - If there are changes to key fields (that is, fields used for matching or for the blocking query) and the update mode is set to pessimistic, potential duplicates are reevaluated for the updated record.
3. If no records are found that match the record's system and local identifier, a second search is performed using the blocking query. A search is performed on each of the defined query blocks to retrieve a candidate pool of potential matches.

Custom plug-in decision point: If `bypassMatching` is set to true, the search steps are bypassed and, if `disallowAdd` is set to false, a new record is added. If `disallowAdd` is set to true, the record is not added and a `MatchResult` object is returned with a result code of 11.

Each record returned from the search is weighted using the fields defined for matching in the inbound message.

4. After the search is performed, the number of resulting records is calculated.
- If a record or records are returned from the search with a matching probability weight above the match threshold, the master index application performs exact match processing (see Step 5).
 - If no matching records are found, the inbound message is treated as a new record. A new EUID is generated and a new record is inserted into the database.
5. If records were found within the high match probability range, exact match processing is performed as follows:
- If only one record is returned from this search with a matching probability that is equal to or greater than the match threshold, additional checking is performed to verify whether the records originated from the same system (see Step 6).
 - If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is set to false, then the record with the highest matching probability is checked against the incoming message to see if they originated from the same system (see Step 6).
 - If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is true, a new EUID is generated and a new record is inserted into the database.
- Custom plug-in decision point:** If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.
- If no record is returned from the database search, or if none of the matching records have a weight in the exact match range, a new EUID is generated and a new record is inserted into the database.

Custom plug-in decision point: If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.

6. When records are checked for same system entries, the master index application tries to retrieve an existing local ID using the system of the new record and the EUID of the record that has the highest match weight.
 - If a local ID is found and same system matching is set to true, a new record is inserted and the two records are considered to be potential duplicates. These records are marked as same system potential duplicates.

Custom plug-in decision point: If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.
 - If a local ID is found and same system matching is set to false, it is assumed that the two records represent the same entity. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 earlier.

Custom plug-in decision point: If `rejectAssumedMatch` is set to true and `disallowAdd` is set to false, a new record is added; if `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11. If `rejectAssumedMatch` and `disallowUpdate` are set to false, the existing record is updated; if `disallowUpdate` is set to true, the update is not performed and a `MatchResult` object is returned with a result code of 13.
 - If no local ID is found, it is assumed that the two records represent the same entity and an assumed match occurs. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 earlier.

Custom plug-in decision point: If `rejectAssumedMatch` is set to true and `disallowAdd` is set to false, a new record is added; if `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11. If `rejectAssumedMatch` and `disallowUpdate` are set to false, the existing record is updated; if `disallowUpdate` is set to true, the update is not performed and a `MatchResult` object is returned with a result code of 13.
7. If a new record is inserted, all records that were returned from the blocking query are weighed against the new record using the matching algorithm. If a record is updated and the update mode is pessimistic, the same occurs for the updated record. If the matching probability weight of a record is greater than or equal to the potential duplicate threshold, the record is flagged as a potential duplicate (for more information about thresholds, see [Sun Master Index Configuration Reference](#)).



Match Types and Field Names

When you create a master index application, you can select a Match Type for each field in the Sun Master Index wizard. Each match type defines a different type of standardization, normalization, phonetic encoding, and matching logic in the `me_fa.xml` file. The following topics describe each match type you can specify in the wizard and how each affects the logic in the `me_fa.xml` file.

- [“Match Types and Standardization Types” on page 133](#)
- [“Sun Match Engine Match Types” on page 134](#)

Match Types and Standardization Types

For each field that will be used for matching in the master index application, you can select a match type in the wizard. When you select a match type for a field, Sun Master Index automatically adds that field to the match string in the `me_fa.xml` file and, in many cases, generates additional fields in the object structure that are not visible on the wizard. These fields are used for searching and matching and they should not be modified.

If new fields are generated, they are automatically incorporated into the configuration files and the database script that creates the master index tables. These fields store standardized, normalized, or phonetic versions of the field, depending on the type of matching you choose. In addition, these fields are assigned a match type in the match string in the `me_fa.xml` file. They might also be defined for standardization in the `me_fa.xml` file, in which case they will also be assigned a standardization type. The types described in this appendix pertain to the Sun Match Engine only.

Note – The match types specified in the `mefa.xml` file for the fields in the match string are not always the same as the match types you specify in the wizard. The match types you can specify from the Configuration Editor more closely match those that are shown in the `mefs.xml` file. This discussion pertains more to the match types on the wizard. For more information, see [Master Index Match Engine Reference](#).

Sun Match Engine Match Types

The Sun Master Index wizard match types for the Sun Match Engine fall into four primary categories.

- “Person Match Types” on page 134
- “BusinessName Match Types” on page 135
- “Address Match Types” on page 136
- “Miscellaneous Match Types” on page 137

The actual standardization and match types entered into the `mefa.xml` file vary for each match type you select in the wizard. The match and standardization types for each type of field are listed in the following descriptions. The match types entered into the `mefa.xml` file correspond to the match types defined in the match configuration file, `MatchConfigFile.cfg`.

Person Match Types

The Person match types include `PersonLastName` and `PersonFirstName`. These match types are used to normalize and phonetically encode name fields for person matching. For each field with one of these match types, the wizard adds two fields to the object structure for phonetic and standardized versions. If you specify a field with a person match type for blocking in the wizard, the phonetic version of the name is automatically added to the blocking query. The following fields are created when you specify one of the Person match types for a field (*field_name* refers to the name of the field specified for Person matching).

- *field_name_Std* – This field contains the normalized version of the name.
- *field_name_Phon* – This field contains the phonetic version of the name.

The corresponding standardization and match types in the `mefa.xml` file are listed in [Table 1](#).

TABLE 1 Person Name Standardization and Match Types

eView Wizard Match Type	Match Field File Standardization Type	mefa.xml Match Type
PersonLastName	PersonName	LastName
PersonFirstName	PersonName	FirstName

BusinessName Match Types

The BusinessName match type is designed to help parse, normalize, and phonetically encode a business name. BusinessName matching adds several fields to the object structure and to the match string. If you specify a business name field for blocking, each parsed business name field is added to the blocking query. The corresponding standardization type in the `me fa.xml` file for all fields selected for BusinessName matching is also BusinessName. The actual match type assigned to each field varies depending on the type of information in each field.

Table 2 lists the fields created when you select the BusinessName match type for a field along with their corresponding match types in the `me fa.xml` file (*field_name* refers to the name of the field selected for BusinessName matching).

Note – Only specify this type of matching for one business name field; otherwise, the wizard will create duplicate entries in the object structure. If more than one field contains the business name, you can add those fields to the standardization structure in the `me fa.xml` file after the wizard creates the configuration files.

TABLE 2 BusinessName Match Types

Field Name	Description	Added to the Match String?	me fa.xml Match Type
<i>field_name</i> _Name	The parsed and normalized version of the business name.	Yes	PrimaryName
<i>field_name</i> _NamePhon	The phonetic version of the business name.	No	
<i>field_name</i> _OrgType	The parsed organization type of the business name.	Yes	OrgTypeKeyword
<i>field_name</i> _AssocType	The association type for the business.	Yes	AssocTypeKeyword
<i>field_name</i> _Industry	The name of the industry for the business.	Yes	IndustryTypeKeyword
<i>field_name</i> _Sector	The name of the industry sector (industries are a subset of sectors).	Yes	IndustrySectorList
<i>field_name</i> _Alias	An alias for the business name.	No	
<i>field_name</i> _Url	The business' web site URL.	Yes	Url

Address Match Types

The Address match type is designed to help parse, normalize, and phonetically encode an address for matching or standardizing address information. Address matching adds several fields to the object structure and to the match string. If you specify an address field for blocking, the parsed fields are added to the blocking query. The corresponding standardization type for fields selected for Address matching is Address. The actual match type assigned to each field varies depending on the type of information in each field.

The fields created when you select the Address match type for a field are listed below along with their corresponding match types in the `mefa.xml` file (*field_name* refers to the name of the field selected for Address matching).

Note – Only specify this type of matching for one street address field; otherwise, the wizard will create duplicate entries in the object structure. If more than one field contains the street address, you can define the additional fields in the standardization structure in the `mefa.xml` file after the wizard creates the configuration files.

TABLE 3 Address Match Types

Field Name	Description	Added to Match String?	mefa.xml Match Type
<i>field_name</i> _HouseNo	The parsed street number of the address.	Yes	HouseNumber
<i>field_name</i> _StDir	The parsed and normalized street direction of the address.	Yes	StreetDir
<i>field_name</i> _StName	The parsed and normalized street name of the address.	Yes	StreetName
<i>field_name</i> _StPhon	The phonetic version of the street name.	No	
<i>field_name</i> _StType	The parsed and normalized street type of the address, such as Boulevard, Street, Drive, and so on.	Yes	StreetType

If you want to search on street addresses but do not want to use these fields for matching, select the Address match type for only one street address field in the wizard. When the wizard is complete, you can remove the address fields from the match string in the `mefa.xml` file.

Miscellaneous Match Types

Several additional match types are defined in the wizard for the Sun Match Engine. These match types are used to indicate matching on a string, date, or number fields other than those described above or to indicate matching on a field that contains a single character (such as the gender field, which might accept “F” for female or “M” for male). These match types do not define standardization for the specified field and do not add any fields to the object structure. If you specify one of these match types for a field in the wizard, the field is added to the match string with a match type of String, Date, Number, or Char.

