Implementing Oracle® Approvals Management

RELEASE 11*i*

May 2004

Table of Contents

1 Introduction to Oracle Approvals Management

Oracle Approvals Management

Oracle Approvals Management (AME) is a self-service Web application that enables users to define business rules governing the process for approving transactions in Oracle Applications where AME has been integrated.

What are the advantages of using Oracle Approvals Management?

Oracle Approvals Management enables business users to specify the approval rules for an application without having to write code or customize the application. Once you define the rules for an application, that application communicates directly with AME to manage the approvals for the application's transactions.

What kind of approval hierarchies are supported?

You can define approvals by job or supervisor hierarchy, or by lists of individuals created either at the time you set up the approval rule or generated dynamically when the rule is invoked. You can link different approval methods together, resulting in an extremely flexible approval process.

Can you use the same rules for different applications?

Yes. You can define rules to be specific to one application or shared between different applications.

How can you ensure that the rules you create are valid?

AME has built-in testing features that enable you to confirm the behavior of new or edited business rules before live execution.

How is a transaction in progress affected by changes in your organization?

Because AME recalculates the chain of approvals after each approval, a transaction is assured to be approved under the latest conditions, regardless of organizational changes, changes to transaction values, rule changes, or currency conversions.

My customer does not have Oracle HR, but has licensed the financials suite and wants to use AME. Can they?

First, customers using any of the financials products but not Oracle HR also install "Shared HR," which is a "lite" version of the HR product that includes the common entities that are needed by all applications. These include organizations, locations, jobs, positions, and people. AME will work with Shared HR. Customers do not need to apply the HR Family Pack if they do not install any of the HRMS applications. They will need to set up the people, jobs, or positions that they want to include in their approval rules.

Second, customers can use AME without either the full HR product or the Shared HR module, by using only FND users as approvers. Such customers would typically create AME approval groups, populate them with FND users, and reference the approval groups in rule using one of the approval-group approval types.

Overview of Oracle Approvals Management

The purpose of Oracle Approvals Management (AME) is to define *approval rules* that determine the approval processes for Oracle applications. Rules are constructed from *conditions* and *approvals*.

Approval Rules

An approval rule associates one or more conditions with an approval in an if-then statement. Each condition tests the value of a variable (known as an *attribute*). The approval defines the list of approvers to which the transaction is routed if the conditions are met:

```
If
   condition C1 is true and
   condition C2 is true and . . .
then
   do approval A1.
```

For example, to create the following approval process:

"For requisitions up to $10,000, require approval from each manager above the requestor up to senior director level."

the rule might be:

```
If
   TRANSACTION_AMOUNT < 10000 USD
then
   require approvals up to at least job level
5
```

You associate a rule with a *transaction type*, which belongs to an application. You can associate the same rule with several transaction types and therefore several applications.

What Happens at Run Time

Once you have defined a set of rules for a transaction type, and the application associated with the transaction type is configured to use AME, the application communicates directly with AME to manage the transaction type's approval processes. Typically the application communicates with AME when a transaction is initiated in the application, and then each time an approver responds to the application's request for approval of the transaction, until all approvers have approved the transaction.

AME records each approval, and recalculates the approver list for a transaction each time an approver responds to a request for approval of the transaction. See How AME Processes Rules at Run Time: page for further details.

The reason why AME recalculates the approver list each time an approver responds is to account for several possible circumstances that can affect a transaction's approver list:

- An attribute value changes, thereby affecting which conditions are true and so which rules apply to the transaction.
- A condition or rule is added, changed, or deleted, again affecting which rules apply to the transaction.
- A change occurs in the organizational hierarchy used by the transaction type's set of rules, thereby changing the membership of the applicable chain of authority.
- Currency exchange rates change, thereby affecting which conditions on currency attributes are true and so which rules apply to the transaction.

By accounting for such changes, AME guarantees that transactions are always approved according to the most current business data possible.

2
Implementing Oracle Approvals Management

Implementing Oracle Approvals Management

To implement AME, you need to carry out the following steps:

Install the Application

AME's installation routines and administration features determine which applications can use AME. Installation and administration are typically jobs for a technical specialist. Installation is generally done only once, and administrative tasks (using AME's Admin tab) are usually only necessary to enable a new application to use AME, or to access or clear a transaction's error log.

> **Note:** Installing AME includes creating a scheduled job that executes the ame_util.purgeOldTempData procedure daily. Failure to perform this task will eventually result in performance degradation and unlimited growth of the size of certain AME database tables.

Assign Users AME ICX Responsibilities and Secured Attributes

AME defines three ICX responsibilities:

- AME Application Administrator
- AME General Business User
- AME Limited Business User

It also defines one secured attribute:

- ame_internal_trans_type_id

("AME" is the Oracle Applications prefix for Oracle Approvals Management.) The Application Administrator responsibility has full access to AME's Web interface. The General Business and Limited Business responsibilities can access the Web interface's Conditions, Groups, Rules, and Test tabs; but not the Attributes, Approvals, or Admin tabs, because these require technical expertise with SQL or Oracle Application internals.

The difference between the General and Limited Business responsibilities is that, while a user with the former responsibility can access any transaction type, a user with the latter responsibility can only access a transaction type if they have been assigned the appropriate ame_internal_trans_type_id secured-attribute value. The value is the transaction type's AME-internal ID. A user with the Application Administrator responsibility can view a transaction type's AME-internal ID by choosing the Admin tab, selecting the Maintain transaction types radio button, and then selecting the transaction type on the list of transaction types. The AME-internal ID can be a negative or positive integer.

Unless otherwise stated, users with General Business and Limited Business responsibilities can access all of the functionality.

Some of the remaining setup steps require the Application Administrator responsibility. If your job is to install, configure, or otherwise administer AME, make sure you have the Application Administrator responsibility before continuing to set up AME.

(Optional) Create Transaction Attributes

In AME, an *attribute* is a named business variable such as TRANSACTION_AMOUNT, whose value AME fetches at run time, when it constructs transactions' approver lists. Only a user with the Application Administrator responsibility can create or alter attributes (using the Attributes tab), because doing so generally requires entering or changing an SQL query.

AME includes the attributes commonly required for the transaction type(s) of each application that can use AME. If your organization has customized an application, or has defined flexfields in it, and wants to use these in the application's approval processes, a user with the AME Application Administrator responsibility must create new attribute names representing the customizations or flexfields, and must define SQL queries that fetch their values at run time. Business users can only select from existing attributes, when they create conditions for AME rules.

Create Conditions

In AME, a *condition* specifies a list or range of attribute values required to make a rule apply to a transaction. For example:

```
USD1000 < TRANSACTION_AMOUNT < USD5000
```

You create and maintain conditions using the Conditions tab.

(Optional) Create Approval Groups

An AME *approval group* is an ordered list of persons and/or user IDs. You can create AME rules to include one or more approval groups in a transaction's approver list. You create and maintain approval groups using the Groups tab. You must create an approval group before using it in an approval-group rule.

Prepare to use the Approval Types

Seeded Approvals Types and Approvals

AME includes a set of seeded approval types and approvals. An *approval* determines which approvers are included in a transaction's approver list. Typically an *approval type* represents a way to ascend a certain organizational hierarchy, including in a transaction's approver list an appropriate chain of authority from the hierarchy; and an approval specifies where the chain starts and ends. If your organization wishes to require approvals from an organizational hierarchy that none of AME's seeded approval types ascend, you need to use a custom approval type. The procedure to create a custom approval type is detailed elsewhere in this guide.

Creating Approval Types

Only a user with System Administrator responsibility can create an approval type, because doing so involves coding, compiling, and testing an approval-type *handler* (a PL/SQL procedure or package) in the APPS schema, and then registering that handler with AME (using the Approvals tab). Like custom attribute and approval-group definitions, custom approval types need only be created once.

Adding Approvals to Existing Approval Types

Even if your organization plans to use AME's seeded approval types, you may need to add to their sets of approvals. For example, the supervisory-level approval type comes with approvals for a supervisory hierarchy having at most 10 levels. If your organization has 15 levels, you need to create supervisory-level approvals for levels 11-15. Once your organization decides which approval types to use, you should compare the seeded approvals with your organization's requirements.

Preparing to use the Job-Level Approval Types

If your organization plans to use one of the job-level approval types, it must first assign a job level to each job defined in HRMS (that is, it must first populate the approval_authority column of the HRMS table per_jobs). Your organization should also have a business process for maintaining job levels. See "Defining a Job" in *Using Oracle HRMS - The Fundamentals* for details.

Define Approval Rules

In AME, an *approval rule* associates one or more conditions with an approval action. The rule applies to a transaction if and only if all of the rule's conditions are true for the transaction.

Each application that can use AME defines one or more *transaction types*. Each transaction type has its own set of approval rules. Several transaction types can share attribute names, while defining separate *usages* for those attribute names. This makes it possible for several transaction types to share conditions and rules. See Attribute Usages: page - 75.

Test Approval Rules

Once a transaction type has a set of rules, it is critical to test the rules, to make sure they apply to the proper cases and do not contain logical gaps or inconsistencies. You can test a set of rules using the Test tab to view the approver list for real transactions, and to create test transactions having combinations of attribute values that represent diverse business cases.

Configure Oracle Applications to use AME

An Oracle Application should be configured to use AME only after thoroughly testing the set(s) of rules defined for that application's transaction type(s) in AME. Consult the application's user or technical documentation to learn how to configure the application to use AME.

3
Attributes

Attributes

An *attribute* is a business variable that has exactly one value for a given transaction. Common attributes are things like a transaction's total amount, a percent discount, an item's category, a person's salary, and so on. In AME, attribute names always appear uppercased, for example TRANSACTION_AMOUNT.

Attribute Types

AME distinguishes five *attribute types*:

- Number
- Date
- String
- Boolean
- Currency

Boolean attributes are either true or false.

Currency attributes differ from the others in having three components to their values: amount, denomination, and conversion method. AME defines a currency attribute type, rather than treating currency values as number attributes, to enable currency conversion when evaluating conditions defined on currency values. You must use the currency attribute type to represent currency variables to take advantage of this functionality.

For example, suppose TRANSACTION_AMOUNT is a currency attribute. At run time, AME might need to evaluate the condition:

```
TRANSACTION_AMOUNT < 500 USD
```

On fetching the attribute's value, AME finds that the amount is in British pounds (not U.S. dollars), and that the attribute requires the Daily conversion method. AME would then use General Ledger's currency-conversion functionality to convert the attribute's value into U.S. dollars, using the Daily conversion method. Having done so, it would evaluate the condition.

Note: To use this functionality, you must enable Oracle General Ledger's currency conversion.

Attribute Usages

All transaction types in AME can share an attribute name, while defining their own method of determining the attribute's value at run time (an *attribute usage*). This makes it possible for several transaction types to share conditions and rules, so that an organization can define a single set of rules that applies to several transaction types (a uniform approvals policy). It also means that an attribute name can be defined in AME, while a given transaction type may not have yet defined a usage for the attribute. A transaction type only has access to conditions defined on attributes for which the transaction type has defined usages. Only users with the System Administrator responsibility can create and edit attribute usages (using the Attributes tab).

There are two kinds of attribute usages: static and dynamic.

Static Attribute Usages

A *static attribute usage* assigns a constant value to an attribute, for a given transaction type. Static usages are common (but not required) for certain mandatory boolean attributes that affect how AME treats all transactions, for example, the AT_LEAST_ONE_RULE_MUST_APPLY attribute (see Mandatory Attributes: page 79). They are similarly used for certain required boolean attributes, for example INCLUDE_ALL_JOB_LEVEL_APPROVERS (see the description of the absolute-job-level approval type under Approval Types for List-Creation and Exception Rules: page - 95 for details).

Syntax Rules for Static Usages

1. Static usages must not use single or double quote marks to demarcate strings.
2. Static usages for boolean attributes must be one of the strings 'true' and 'false'.
3. Static usages for number attributes must be either an integer or a decimal number in decimal (not scientific) notation. For example, '1' and '-2' are acceptable integer values, and '-3.1415' is an acceptable decimal value.
4. Static usages for date attributes must use the format model ame_util.versionDateFormatModel, whose value is:

   ```
   YYYY:MON:DD:HH24:MI:SS
   ```

   For example, '2001:JAN:01:06:00:00' is an acceptable date value.

   Note: This format model differs slightly from the "canonical" format model, which contains a space character. Space characters are problematic for variable values passed to a Web server via the HTTP GET method.
5. Static usages for string attributes can be any text value that fits in a varchar2 of length ame_util. attributeValueTypeLength, which is currently 100. (You may wish to check your AME installation's source code to verify that this constant still has the same value.). The text value may include spaces and ordinary punctuation. It is case-sensitive.
6. Static usages for currency attributes must have the form:

```
amount,code,type
```

where *code* is a valid currency code and *type* is a valid currency-conversion type. There should be no space characters other than those in the code and type values. The amount should use a period, not a comma, as a decimal point (if any). For example:

```
5000.00,USD,Corporate
```

is a valid currency value.

7. Static usages may be null. To create a null static usage, leave the usage field empty on the Create an Attribute, Edit an Attribute, or Mandatory Attribute Query Entry page.

Dynamic Attribute Usages

A *dynamic attribute usage* assigns an SQL query to an attribute, for a given transaction type. The query must follow certain syntax rules. AME executes the query at run time to determine the attribute's value. Dynamic usages are common for all attributes other than the two classes of boolean attributes described above.

The execution of dynamic attribute usages' queries represents the majority of AME's runtime overhead. Therefore, optimizing your dynamic-usage queries can have a big impact on AME's performance. Make sure you optimize these queries thoroughly, especially if the transaction type that owns them processes a high volume of transactions.

Syntax Rules for Dynamic-usage Queries

1. The query must fit in the column ame_attribute_usages.query_string, which is a varchar2(2000). If your query is long, you may wish to compare its length with the current table definition in your applications instance. (You can avoid the length constraint by encapsulating your query in a function that you compile on the database, and then selecting the function's value from dual in your query.)
2. The queries for all data types other than currency must select one column; queries for the currency data type must select three columns.
3. Each selected column must convert to a value that fits in a varchar2 of length ame_util. attributeValueTypeLength, which is currently 100. (You may wish to check your AME installation's source code to verify that this constant still has the same value.)
4. Queries for boolean attributes must select one of two possible values, ame_util.booleanAttributeTrue and ame_util.booleanAttributeFalse.
5. Queries for date attributes must *c*onvert a date value into a varchar2 using the ame_util.versionDateToString function, to guarantee that AME stores the date value using the ame_util.versionDateFormatModel. AME can only evaluate conditions defined on a date attribute correctly when that

attribute’s dynamic usage converts the attribute’s date value using this format model, because AME stores the date as a varchar2, and attempts to convert the stored value back to a date using the same format model. For example:

```
   Select
ame_util.versionDateToString(sysdate) from
dual
```

is a correct dynamic usage for a date attribute.

6. Queries for number and currency attributes must select the number or currency amount converted to a varchar2 by:

   ```
   fnd_number.number_to_canonical.
   ```

7. Queries for header-level attributes may (but are not required to) contain the transaction-ID placeholder ame_util.transactionIdPlaceholder, which is ’:transactionId’. The transaction-ID placeholder is a true dynamic PL/SQL bind variable. That is, at run time, AME binds a transaction-ID value to this variable before dynamically executing the query. A condition of a where clause referencing this bind variable must have the form:

   ```
   transaction ID = :transactionId
   ```

   where *transaction ID* is a column that contains the transaction ID passed to AME at run time by the application whose transaction type uses the query. Queries for line-item-level attributes may also (but are not required to) contain the line-item-ID placeholder ame_util. lineItemIdPlaceholder, which is ’:lineItemIdList’. The line-item-ID placeholder is not a true bind variable; it is merely a placeholder for which AME substitutes a line-item-ID list such as ”(‘1’, ‘2’, ‘3’)”, before parsing the query. A condition of a where clause referencing this placeholder must have the form:

   ```
   line-item ID in :lineItemIdList
   ```

   where *line-item ID* is a column that contains the transaction type’s line-item IDs. If a query includes the line-item-ID placeholder, it must also include an order-by clause of the form:

   ```
   order by line-item ID
   ```

   (That is, the query must order the attribute values in ascending order of the associated line-item IDs.) AME will not let you save a query referencing the line-item-ID placeholder that does not contain an order-by clause. If your order-by clause does not reference the same (line-item-ID) column as the where-clause condition that references the line-item-ID placeholder, AME will raise an exception at run time.

This sample query satisfies the above rules, for a line-item currency attribute:

```
select
  fnd_number.number_to_canonical(amount),
  currency_code,
  currency_conversion_type
```

```
from some_app_line_items
where
  header_id = :transactionId and
  line_item_id in :lineItemIdList
order by line_item_id
```

Attribute Classifications

AME classifies attributes not only according to their data type, but also according to whether they are mandatory, required, and active.

Mandatory Attributes

An attribute is *mandatory* if every transaction type must define a usage for it. AME uses mandatory attributes in its internal logic, and so fetches their values for all transactions. A transaction type does not need to define usages for non-mandatory attributes. All attributes that an end-user defines are non-mandatory. AME also includes some non-mandatory attributes that are commonly used by various transaction types.

AME's mandatory attributes are:

ALLOW_DELETING_RULE_GENERATED_APPROVERS

This is a boolean attribute that determines whether AME allows an application to delete from a transaction's approver list (at run time) approvers required by the appropriate transaction type's rules.

ALLOW_REQUESTOR_APPROVAL

This is a boolean attribute whose value determines whether AME lets a requestor approve their own transaction, if they have sufficient signing authority.

AT_LEAST_ONE_RULE_MUST_APPLY

This is a boolean attribute determining whether AME raises an exception when no rules apply to a transaction at run time.

EFFECTIVE_RULE_DATE

This is the date that determines which rules are *active* for a given transaction, in the following sense. When AME begins to process a transaction, it first determines which rules have start dates that precede the effective date; and that have end dates which are either null or which follow the effective date. These rules are active for the transaction. AME then evaluates each active rule's conditions to see whether the rule actually applies to the transaction.

For most transaction types, sysdate is the appropriate EFFECTIVE_RULE_DATE value. To use this value, give EFFECTIVE_RULE_DATE a static null (empty) usage. (This will be more efficient at run time than giving it a dynamic usage that selects sysdate from dual.)

EVALUATE_PRIORITIES_PER_LINE_ITEM

Relative rule priorities can be evaluated per transaction or line item. This attribute determines the time of evaluation that the engine performs. This attribute is only significant if the USE_RESTRICTIVE_LINE_ITEM_EVALUATION is set to true. If it is set to not true (i.e. false or null), priority evaluation is always per transaction, regardless of the value of EVALUATE_PRIORITIES_PER_LINE_ITEM. (For this reason the attributes tab prevents one from setting EVALUATE_PRIORITIES_PER_LINE_ITEM to a static-true value if USE_STRICT_LINE_ITEM_EVALUATION is static and not true.) . For further information on rule priorities please review the rules and administration sections. This attribute will only appear in the list (of mandatory attributes) if the transaction type contains a line-item-ID query string.

TRANSACTION_DATE

This is the date a transaction was first submitted. Note that AME always fetches a transaction's *current* attribute values, and then applies the appropriate transaction type's rules active as of EFFECTIVE_RULE_DATE, to (re)generate the transaction's approver list. It does *not* use the attribute values or rules existing at the transaction date. You may however wish to sort your transactions according to, for example, the financial-reporting period in which they originate, in your rules. This attribute gives you a convenient way to do so.

TRANSACTION_GROUP_ID

This is the ID of the business group in which a transaction originates. (Throughout AME, a business-group ID is a value from the business_group_id column of the HRMS table per_business_groups.) This attribute may have a null value at run time. AME uses this attribute to evaluate rule constraints at run time (see Constraints: page - 133).

TRANSACTION_ORG_ID

This is the ID of the organization in which a transaction originates. (Throughout AME, an organization ID is a value from the organization_id column of the HRMS table hr_all_organization_units.) This attribute may have a null value at run time. AME uses this attribute to evaluate rule constraints at run time (see Constraints: page - 133).

TRANSACTION_REQUESTOR_PERSON_ID

This is the person ID of the individual submitting a transaction. (Throughout AME, a person ID is a value from the person_id column of the HRMS view per_all_people_f.) This attribute may have a null value at run time (when, for example, a generic user account is used to submit a transaction). When a given transaction type has the possibility of a null-valued requestor person ID, you should avoid using the job-level approval types (see Approval Types: page - 94).

TRANSACTION_REQUESTOR_USER_ID

This is the user ID of the user submitting a transaction. This attribute may have a null value at run time.

TRANSACTION_SET_OF_BOOKS_ID

This is the ID of the set of books that owns a transaction. (Throughout AME, a set-of-books ID is a value from the set_of_books_id column of the General Ledger table gl_sets_of_books.) This attribute may have a null value at run time. AME uses this attribute to evaluate rule constraints at run time (see Constraints: page - 133).

USE_RESTRICTIVE_LINE_ITEM_EVALUATION

This is a boolean attribute indicating whether at runtime AME requires that a single line item satisfy all line-item conditions in a given rule, for the rule to apply to the transaction being processed. If the attribute is true, a rule containing line-item conditions applies to a transaction only if one of the transaction's line items satisfies all of the rule's line-item conditions. If the attribute is false, different line items may satisfy different line-item conditions.   This attribute will only appear in the list (of mandatory attributes) if the transaction type contains a line-item-ID query string.

WORKFLOW_ITEM_KEY

This is the item key (typically the transaction ID) of the transaction, when the transaction gets processed at run time by a workflow in the application that owns the transaction type. (If the application does not use Workflow, this attribute's value will be null.) AME only uses this attribute to communicate exception-related information to Workflow at run time. You should not use this attribute in your conditions.

WORKFLOW_ITEM_TYPE

This is the item type of the workflow that calls AME's application-programming interface (API) at run time to process a given transaction type's transactions. (If the application that owns the transaction type does not use Workflow, this attribute's value will be null.) AME only uses this attribute to communicate exception-related information to Workflow at run time. You should not use this attribute in your conditions.

Required Attributes

An attribute may be *required* by an approval type, for the approval type to operate at run time. In this case, AME does not let you create a rule using the approval type when your transaction type has not yet defined a usage for the approval type's required attributes.

Active Attributes

An attribute is *active for a rule* if one or more of the rule's conditions is defined on the attribute, or if the approval type of the rule's approval requires the attribute. An attribute is *active for a transaction type* if the attribute is mandatory, or if it is active for at least one of the transaction type's rules. AME only fetches the values of active attributes at run time, and it only lists a transaction's active attributes when you create a test transaction on the Test tab.

Attribute Levels

Attributes divide further according to their level. *Header-level attributes* relate to an entire transaction. This is the common case. Mandatory attributes, for instance, are always header-level attributes. *Line-item-level attributes* have distinct values for each of a transaction's line items. For example, an expense report might have line items, each representing a distinct expenditure being reported; and the Web Expenses transaction type might define a LINE_ITEM_AMOUNT currency attribute to represent the cost of each expenditure.

A transaction type might not have line items. Even if the application that owns a transaction type does have line items, the transaction type might not enable line-item-attribute functionality in AME. (To enable the line-item functionality for a given transaction type, someone with Application Administrator responsibility must enter a line-item-ID query for the transaction type on the Admin tab. See Administration: page - 112 for details.)

How does AME use Attributes?

When AME starts to calculate a transaction’s approver list at run time, the first thing it does is fetch the values of each attribute that is active for the transaction type. (To do this, AME either fetches the constant value assigned to the attribute, or fetches the attribute’s query and then executes it.) If an attribute having a dynamic usage is a header-level attribute, AME executes the query once for the entire transaction. If the attribute is a line-item-level attribute, AME executes the query once for each of the transaction’s line items.

After fetching all of the active attributes’ values, AME checks whether each of a transaction type’s rules applies to the transaction. It does this by determining whether each of the rule’s conditions is true. For conditions defined on header-level attributes, the condition is true if the attribute’s value lies within the list or range of values that the condition defines. For conditions defined on line-item-level attributes, the condition is true if the value of the attribute for any one line item lies within the list or range of values that the condition defines.

Maintaining Attributes

You can view, create, edit and delete attributes using the Attributes tab. This tab is available only to users with the Application Administrator responsibility.

If you have the General or Limited Business responsibility, AME presents you with an appropriate attribute list whenever you need to select an attribute in the process of creating a condition or rule. If you need to have a custom attribute created or edited, you must ask a user with the Application Administrator responsibility to perform that task.

" **To display the list of attributes for a transaction type:**

1. Choose the Attributes tab.
2. Select the transaction type whose attributes you want to view, then choose the Continue button.

The attributes list appears. The list is subdivided into mandatory attributes, non-mandatory header attributes, and non-mandatory line-item attributes.

Note: If the transaction type does not have a line item ID query string defined, references to attribute level in the following text should be ignored.

" **To create an attribute:**

1. Display the list of attributes.
2. Choose the Add Attribute button. This starts the Create an Attribute wizard.
3. Select the attribute's level (header or line-item) if the transaction type has enabled line-item attributes.
4. Select a pre-existing attribute name, or enter a new attribute name. If you plan to create one or more rules referencing the attribute, and to share the rule(s) across several transaction types, you just need to create the attribute's name for the first transaction type, and then select it from the list of shareable attribute names for all remaining transaction types. (You must enter a distinct usage for the attribute name, for each transaction type.)

   All attribute names, including those you create, are shareable; so make sure that your attribute name's degree of generality reflects your intentions. For example, if you want to create an attribute specific to the Web Expenses transaction type, you might begin your attribute name with the prefix 'WEB_EXPENSES_'.

   Note: If you enter an attribute name in lower or mixed case, AME changes it to all upper case when it saves your work.

   The names of attributes whose values will be person IDs or user IDs should end in 'PERSON_ID' or 'USER_ID', respectively; for example, 'HR_MANAGER_PERSON_ID' and 'EMPLOYEE_STOCK_ANALYST_USER_ID'. This convention signals the Test tab's test-transaction functionality to let the end user query for persons and users by name, and then to display a user-friendly description of the person or account selected by the end user.

5. Select an attribute type. Remember to use the currency type for attributes reflecting monetary values.
6. Select the radio button indicating the type of attribute usage you want to use.
7. Enter the attribute usage. (See Attribute Usages: page 75 for details of the syntax rules for attribute usages.)
8. Choose the Create Attribute button.

Your new attribute appears in the appropriate section of the attributes list.

" **To edit an attribute:**

1. Display the list of attributes.
2. Select the name of the attribute that you want to edit.
3. Make your changes on the Edit an Attribute page.
4. Choose the Submit Changes button to save your changes.

When you are satisfied with your changes, you can choose the Quit button to return to the attribute list.

When you change an attribute's name, type, or description, your changes apply for all transaction types. Changes to attribute usages only apply to the transaction type for which you make the change.

You cannot change the name, type, or description of seeded attributes. You must instead create a new attribute with the name, type, and description that you want. In some cases, you can create or edit a seeded attribute's usage. (If the usage is editable, it will appear that way on the Edit an Attribute page.)

If you change an attribute's type, make sure you also change its usage (for every transaction type that uses the attribute name) to provide data of the appropriate type. If you change a usage's type from static to dynamic, make sure you change the usage accordingly.

" **To delete an attribute:**

1. Display the list of attributes.
2. Select the check box next to the attribute name, in the Delete column.
3. Choose the Delete Checked Attributes button.
4. Confirm the deletion when prompted.

You can delete several attributes at once.

Note: You cannot delete seeded attributes.

Can a dynamic attribute usage reference the value of another attribute?

For example:

Attribute1: select column1 from table1 where column2 = :transactionId
Attribute2: select column3 from table2 where column4 = :Attribute1

Dynamic attribute usages cannot reference other attributes' values. To implement the above example, the second attribute's dynamic usage would be:

select column3 from table2 where column4 =
(select column1 from table1 where column2 = :transactionId)

AME does not allow references to attribute values within dynamic usages for two reasons:

1. It is not practical for AME to guarantee that a transaction type's active attributes (those attributes whose values the engine and/or rules require to determine which rules apply to a transaction of that type) will always be fetched in a fixed order. As a result, AME may not have fetched the value of Attribute1 before it tries to fetch the value of attribute2. Even if AME were to guarantee an attribute-fetching order, that would not solve the general problem. It would only solve the problem for attributes that come after the ones they depend on, in the order.

2. Were AME to allow this sort of dependency among attributes, cycles could occur. Attribute1 could depend on Attribute2, and Attribute2 could depend on Attribute3, which in turn could depend on Attribute1. There would be no way in logic to resolve these attributes' values.

4
Conditions

Conditions

An approval rule's "if" part consists of one or more conditions, each of which is either true or false. For the rule to apply to a transaction, all of its conditions must be true for the transaction.

Condition Types

There are three types of conditions:

- Ordinary
- Exception
- List-modification

Ordinary and *exception* conditions associate an attribute with a list or range of possible attribute values. If at run time the attribute has one of these values, the condition is true; otherwise it is false. All six rule types can have one or more ordinary conditions, and most rules have at least one.

The differences between ordinary and exception conditions lie in how the two condition types are used. Ordinary conditions can occur in all rule types, but exception conditions can only occur in exception rules. Also, AME does not compare the attributes in an exception's exception conditions with those in a list-creation rule's ordinary conditions, to determine whether the exception suppresses the list-creation rule. (See List-Creation Exceptions: page - 130 for details.)

Ordinary and exception conditions have one of three forms. Conditions on date, number, and currency attributes have the form:

```
lower limit <{=} attribute name <{=} upper
limit
```

Conditions on boolean attributes have the form:

```
attribute name is {true, false}
```

Conditions on string attributes have the form:

```
attribute name in {value 1, value 2, . . . }
```

When you create or edit a condition of the first form, you decide whether to include each equals sign in the condition by selecting the appropriate values for the Include Lower Limit and Include Upper Limit radio buttons. If you create a set of conditions of this form on the same attribute, and the conditions have successive ranges of values, you generally should only include the value between each successive pair in one of the pair. For example, the conditions:

```
1,000 <= TRANSACTION_AMOUNT < 2,000
2,000 <= TRANSACTION_AMOUNT < 3,000
```

have successive ranges of values that avoid overlapping at 2,000. In this way, at most one of the conditions will ever be true for any given transaction, which is typically the desired behavior.

A *list-modification* condition identifies an approver that might appear in a transaction's chain of authority. When a list-modification or substitution rule containing the list-modification condition applies to a transaction, the rule's action is applied to the approver that the condition identifies.

Maintaining Conditions

You can view, create, edit and delete conditions using the Conditions tab.

" **To display the list of conditions:**

1. Choose the Conditions tab.
2. Select the transaction type, then choose the Continue button.

The conditions list appears, displaying the conditions in three sections according to condition type.

Conditions are shared

The conditions list contains all conditions defined on attributes for which the selected transaction type has defined usages. Transaction types *always* share conditions, regardless of the transaction type you select before creating a condition. This sharing makes it possible for several transaction types to share rules. It also means that when you edit or delete a condition, the change applies to all transaction types. To help you avoid unintentionally changing or deleting a condition used by several transaction types, the conditions list flags conditions used by several transaction types with an asterisk.

" **To create a condition:**

1. Display the list of conditions.
2. Choose the Add a Condition button. This starts the Create a Condition wizard. This wizard guides you through the following steps, for ordinary and exception conditions:
3. Select a condition type. If the transaction type has enabled line-item attributes, you must select whether to create an ordinary or exception condition on a header-level or line-item-level attribute. Most of the time, you will create an ordinary condition (or an ordinary condition on a header-level attribute). Make sure you understand how the rule types (see Rule Types: page - 129) use different condition types, before deciding which type of condition to create.
4. Select the condition's attribute.

The rest of the wizard depends on the attribute's type. For conditions on boolean attributes:

5. Select the *attribute* value ('true' or 'false') that makes the *condition* true. Make sure not to confuse the boolean values of the attribute and condition. A condition on a boolean attribute can be defined so that the *condition* is true when the *attribute* is false.

For conditions on date, number, and currency attributes:

6. Enter or select the condition's lower and upper limits.

7. Select whether to include each limit in the range of values making the condition true. (Including a limit is equivalent to including an equals sign, as discussed above.)
8. For conditions on currency attributes, select the currency code representing the denomination of the lower and upper limits. (See How does the Euro Affect Conditions on Currency Attributes?: page 89 to make sure you avoid using certain pre-euro currency codes.)

For conditions on string attributes:

9. Enter a text value that makes the condition true.

   Note: Text values are case-sensitive.
10. (Optional) Choose the Create Text Value button to add another text value. On the resulting Add a Text Value page, enter the value and choose the Add Text Value button. Repeat this process as long as you want to add more text values.

You can now choose the Quit button to return to the conditions list. Your new condition will appear in the appropriate sub list of the conditions list.

" **To edit a condition:**

1. Display the list of conditions.
2. Select the condition you want to edit. The condition-editing wizards closely parallel the condition-creation wizards described above.

" **To delete a condition:**

1. Display the list of conditions, ensuring that you select a transaction type that has defined a usage for the attribute used by the condition you want to delete.
2. Select the check box next to the condition, in the Delete column.
3. Choose the Delete Checked Conditions button.
4. Confirm the deletion when prompted.

You can delete several conditions at once.

5 Approvals

Approvals

An *approval* is an instruction to AME to include a given set of approvers in a transaction's approver list. Approvals constitute the "then" part of approval rules. When a rule applies to a transaction, AME processes the rule's approval to add the appropriate set of approvers to the transaction's approver list. Approvals have parameters and descriptions, and are of a given approval type.

Approval Parameters

An approval's parameter represents formally the approval's instruction, for a PL/SQL package or procedure known as the *handler* for the approval's type (see Approval Types). Each approval type has its own approval-parameter syntax rules (see the appropriate approval type for the approval type's parameter-syntax and -semantics rules). You must adhere carefully to the syntax rules when you create a new approval for a seeded approval type. (You can check your understanding of an approval type's syntax requirements by reviewing the parameters of some of the approval type's approvals, using the Approvals tab.)

Approval Descriptions

An approval's description should be a complete imperative sentence expressing the approval's requirements. When you create an approval for a seeded approval type, the new approval's description should resemble as much as possible those of existing approvals of the same type. For example, a supervisory-level approval that requires the default chain of authority to ascend the supervisory hierarchy 15 approvers should have the description:

```
Require approvals up to the first 15
superiors.
```

Approval Types

An approval is always of a given approval type. An *approval type* represents a way to calculate which approvers to add to a transaction's approver list. For example, the absolute-job-level approval type ascends a managerial chain of authority (usually starting from the supervisor of a transaction's requestor), adding each manager in the chain to the transaction's default chain of authority, until it reaches a manager having at least, or at most, a certain job level (depending on the action).

AME includes a set of seeded approval types, many of which enable you to ascend commonly used organizational hierarchies. If none of the seeded approval types meets your organization's requirements, you need a custom approval type. (The most common reason to create a custom approval type is to represent a particular organizational hierarchy.) An AME user with the Application Administrator responsibility must use the Approvals tab to define a new approval type. (See How to Create a Custom Approval Type: page - 140 for the technical details.)

Every approval type specifies a way to identify an approver's *surrogate*. (Typically the surrogate is just the approver's supervisor.) AME substitutes the surrogate for the approver in a transaction's approver list if the application in which the transaction originates flags the approver as unresponsive. (Whether or when an application does this varies with the application. Consult the application's user documentation for details.)

Required Attributes

Certain approval types require that a transaction type define usages for one or more attributes, to use approvals of that type in its rules. This occurs when the approval type uses the attributes' values to calculate which approvers to include in a transaction's approver list. The Approvals tab lists each approval type's required attributes.

Note: If a seeded attribute is already a mandatory attribute, AME does not include it among an approval type's required attributes, even if the approval type uses the attribute's value at run time.

Approval Types for List-Creation and Exception Rules

Use the following seeded approval types in list-creation and exception rules:

Absolute job level

The absolute-job-level approval type generates a default chain of authority by ascending the supervisory hierarchy and including in a transaction's approver list each person found in the ascent. (The supervisory hierarchy is defined by the HRMS per_all_assignments_f, per_jobs, and per_all_people_f tables and views. To see exactly how AME ascends the hierarchy, please review the source code for the ame_absolute_job_level_handler PL/SQL package.) Where the ascent starts depends on the value of a required attribute. How high in the hierarchy the ascent goes depends on the particular approval, and on the value of another required attribute.

By default, the ascent starts with the supervisor of the transaction's requestor, whom the mandatory attribute:

```
TRANSACTION_REQUESTOR_PERSON_ID
```

identifies. If the required attribute:

```
JOB_LEVEL_NON_DEFAULT_STARTING_POINT_
PERSON_ID
```

has a non-null value at run time, the ascent starts instead with the person whom this attribute identifies. You can instead cause the ascent sometimes to start with someone besides the transaction requestor. To do this, compile a function that returns a non-null person ID only for the desired set of transactions, and null otherwise. Then have the above attribute's (dynamic) usage select the function from dual.

The absolute-job-level approval type's ascent up the supervisory hierarchy stops when it reaches one or more approvers having a sufficient job level. (A *job level* is a value in the authority_level column of the HRMS table per_jobs.) The particular job level required varies with the approval (AME seeds job levels one through 10).

Absolute job-level approvals can also require up to *at least* a given job level or *at most* a given job level. For example, if the rules require approvals up to at least job level seven, and the hierarchy above the requestor skips from job level six to job level eight, the default chain of approval will include the approver at job level eight. If the rules required approvals up to *at most* job level seven in these circumstances, the default chain will only include the approver with job level six.

Two job-level approvals may combine in an interesting way. Suppose, for example, that one rule requires approvals up to at least level five, and another requires approvals up to at most level six. If the hierarchy skips from job level four to job level seven, AME can only satisfy both rules by including in the chain of authority the approver with job level seven. In this case, the at-least approval is more stringent than the at-most approval, even though the at-most approval nominally requires a higher job level. Remember that AME always satisfies the most stringent rule among those applicable; and be aware that for the job-level approval types, which rule is most stringent can vary as the foregoing example illustrates.

It sometimes happens that several consecutive approvers in an ascent up the supervisory hierarchy have the same job level. If this is the highest job level required by the rules, whether AME includes all of the approvers or only the first depends on the value of the required boolean attribute:

```
INCLUDE_ALL_JOB_LEVEL_APPROVERS
```

The parameters of absolute-job-level actions have the syntax:

```
n{+,-}
```

where *n* is a positive integer. The parameter is interpreted to require ascending the supervisory hierarchy to at least *n* job levels if a plus sign is present, or to at most *n* job levels if a minus sign is present. For example, the approval described as, "Require approvals up to at least level 1." has the parameter '1+'.

Required attributes: INCLUDE_ALL_JOB_LEVEL_APPROVERS, JOB_LEVEL_NON_DEFAULT_STARTING_POINT_PERSON_ID

Manager then Final Approver

The manager-then-final-approver approval type is an absolute-job-level variant. Instead of requiring approval from every person in an ascent up the hierarchy, this approval type only includes the first and last approvers in the ascent. Note that this and other variants on the absolute-job-level approval type all require the same attributes as the absolute-job-level type requires, and treat them the same way. The syntax rules for manager-then-final-approver approvals are the same as those for absolute-job-level approvals.

Required attributes: INCLUDE_ALL_JOB_LEVEL_APPROVERS, JOB_LEVEL_NON_DEFAULT_STARTING_POINT_PERSON_ID

Final Approver Only

The final-approver-only approval type is another absolute-job-level variant. It only includes the last approver in the ascent up the supervisory hierarchy. The syntax rules for final-approver-only approvals are the same as those for absolute-job-level approvals.

Required attributes: JOB_LEVEL_NON_DEFAULT_STARTING_POINT_PERSON_ID

Relative Job Level

The relative-job-level approval type is a third absolute-job-level variant. However, its actions specify an ascent up the supervisory hierarchy some number of job levels beyond that of the requestor. The syntax rules for relative-job-level approvals are the same as those for absolute-job-level approvals. Note however that the integer *n* is here interpreted relative to the requestor's job level.

Required attributes: INCLUDE_ALL_JOB_LEVEL_APPROVERS, JOB_LEVEL_NON_DEFAULT_STARTING_POINT_PERSON_ID

Line-Item Job-Level Chains of Authority

Generates a sub chain of authority for each line item of a transaction having line items. Ascends the HR supervisor hierarchy in the same way as absolute job level approval type.

Required attributes:
INCLUDE_ALL_JOB_LEVEL_APPROVALS,
LINE_ITEM_STARTING_POINT_PERSON_ID

Supervisory Level

The supervisory-level approval type ascends the same supervisory hierarchy as the job-level approval types. However, its actions specify the number of supervisors, rather than job levels, to ascend. (There is no precise correlation between the number of supervisors ascended and the job level. As the discussion above of the absolute-job-level approval type explains, a job level can be skipped, or can occur several times in a row.) The parameters of supervisory-level approvals are positive integers indicating the number of supervisors to ascend.

There are two required attributes.
SUPERVISORY_NON_DEFAULT_STARTING_POINT_PERSON_ID This attribute's value can be null. If the value is null for a given transaction, the transaction's chain of authority will start with the supervisor of the person identified by TRANSACTION_REQUESTOR_PERSON_ID. If the value is non-null, it must identify by person ID the person who should be the first approver in the transaction's chain of authority.

TOP_SUPERVISOR_PERSON_ID. This attribute's value can be null. When it is null, AME raises an exception if it does not find a supervisor while ascending the supervisory hierarchy. When the value is non-null, if AME does not find a supervisor while ascending the hierarchy, it checks whether the last supervisor is the person identified by the attribute value. If so, AME ends the chain of authority with this person, without raising an exception. If not, AME raises an exception.

Required attributes:
SUPERVISORY_NON_DEFAULT_STARTING_POINT_PERSON_ID, TOP_SUPERVISOR_PERSON_ID

Dual Chains of Authority

A class of transactions may require approval from two chains of authority, each starting at a different point in the supervisory hierarchy. (For example, an employee transfer may require approval from chains of authority starting at the employee's old and new supervisors.) In such cases, use the dual-chains-of-authority approval type. This approval type puts one chain of authority after the other, in a transaction's approver list.

The dual-chains-of-authority approval type requires two attributes:

```
FIRST_STARTING_POINT_PERSON_ID
SECOND_STARTING_POINT_PERSON_ID
```

These attributes' values identify the first approver in each chain of authority generated by the dual-chains-of-authority approval type, so they must always have non-null values.

The dual-chains-of-authority approval type differs from the other seeded approval types. In this case, *at least two* rules using the approval type must apply to *each* transaction to which any such rule applies—at least one such rule per subchain. This is necessary because dual-chains-of-authority approvals can represent (absolute- or relative-job-level) requirements for only one of the subchains. The safest way to make sure at least two dual-chains rules always apply to any transaction to which one applies is to define pairs of rules with identical conditions and different approvals, one for each subchain. For example, you might define the following pair of rules for the Human Resource Self-Service transaction type:

Rule G

```
If
   TRANSACTION_CATEGORY in {TRANSFER}
then
   Require approvals at most 3 levels up in
the first chain
```

Rule H

```
If
   TRANSACTION_CATEGORY in {TRANSFER}
then
   Require approvals at least 2 levels up in
the second chain.
```

You may include several rules for each subchain. When you do, AME determines which has the most stringent requirement for that subchain, and enforces that requirement.

The parameters of dual-chains-of-authority approvals have the syntax:

```
{1,2}{A,R}n{+,-}
```

The first integer (a one or a two) indicates the (first or second) subchain. The letter (an 'A' or an 'R') indicates whether *n* (which must be a positive integer) should be interpreted as an absolute or a relative job level. The plus or minus sign is interpreted as for the job-level approval types. For example, the approval described as, "Require approvals at most 8 levels up in the second chain." has the parameter '2R8-'.

Required attributes: INCLUDE_ALL_JOB_LEVEL_APPROVERS, FIRST_STARTING_POINT_PERSON_ID, SECOND_STARTING_POINT_PERSON_ID

Approval Types for List-Modification Rules

Use the following approval types in list-modification rules:

Non-Final Authority

The non-final-authority approval type extends a chain of authority beyond the approver identified in a list-modification's list-modification condition, up to some absolute or relative job level, depending on the particular approval's requirement. Note that the list-modification condition must identify an approver as a *person* (by person ID), and not as a *user account* (user ID), so that this approval type can make the necessary job-level calculations. (User accounts are not assigned job levels.)

Non-final authority approvals' parameters have the syntax:

```
{A,R}n{+,-}
```

The 'A' or 'R' indicates whether *n* (a positive integer) represents an absolute- or relative-job-level requirement. The plus or minus sign is interpreted as for the job-level approval types. For example, the approval described as, "Require approvals at most 2 levels up." has the parameter 'R2-'.

Final Authority

The final-authority approval type is unique in that it has only one approval (and that approval has a null parameter). A rule using this approval truncates the default chain of authority after the approver identified by the rule's list-modification condition, thereby granting that approver signing authority, regardless of their position in any organizational hierarchy. (If the target approver forwards without approving, this approval type truncates the chain of authority after the last forwardee following the target approver.)

You can use the final-authority approval to grant signing authority to both persons and user accounts, as long as the application that owns the transaction type that in turn owns the rules can include user accounts in its approver lists. (All applications support person approvers, but not all applications support user-account approvers. Web Expenses, for example, currently does not.)

Approval Types for Substitution Rules

There is only one seeded approval type for substitution rules:

Substitution

The substitution approval type replaces the approver identified by a substitution rule's list-modification condition with another approver. You can use this approval type to substitute persons and user accounts interchangeably, as long as the application that owns the transaction type that in turn owns the rules can include user accounts in its approver lists. (All applications support person approvers, but not all applications support user-account approvers. Web Expenses, for example, currently does not.)

The parameters for substitution approvals have the following syntax:

```
{'user_id', 'person_id'}:id
```

For example:

```
'user_id:123' and 'person_id:123'
```

are both valid substitution parameters. However, the AME user interface enables you to select an approver type and then query for an approver, to create or edit a substitution approval. Once you do, AME generates the approval description automatically. You do not edit the parameter or description directly.

Approval Types for Approval-Group Rules

Use the following approval types for pre- and post-approval rules:

Pre-Chain-of-Authority Approvals

The pre-chain-of-authority-approvals approvals type adds an approval group to a transaction's approver list *before* all approvers in the transaction's chain(s) of authority. The members of the approval group appear in the approver list in the order defined by the approval group. Pre-chain-of-authority-approvals approvals' parameters are the ame_approval_groups.approval_group_id value corresponding to the approval group of interest. The AME Web interface lets you select the approval group from a user-friendly list when you create a pre-chain-of-authority-approvals approval, so you do not have to query for the group's ID. The approval group can either contain a static list of members or can contain an SQL query that dynamically fetches the approvers. If the possibility arises where the approval group can contain no members the required attribute, ALLOW_EMPTY_APPROVAL_GROUPS should be set to true. If this is set to false, AME will raise an exception if the group is empty.

Required attributes: ALLOW_EMPTY_APPROVAL_GROUPS

Post-Chain-of-Authority Approvals

The post-chain-of-authority-approvals approvals type mimics the pre-chain-of-authority-approvals type, but it inserts approvers after all authority approvers.

Required attributes: ALLOW_EMPTY_APPROVAL_GROUPS

Approval-Group Chain of Authority

Sometimes it is required to include an approval group into the chain of authority that is generated when the rules are evaluated. The chain may be included within the line item chain or the header chain.

Required attributes: ALLOW_EMPTY_APPROVAL_GROUPS

Dynamic Pre-Approver (DEPRECIATED – Use Dynamic Groups)

The dynamic-pre-approver approval type inserts a single pre-approver (not an approval group) identified by an attribute at run time. This approval type has no seeded approvals; you must create them before using the approval type.

A dynamic-pre-approver approval's parameter must be a string attribute's name. At run time, the attribute's value must be a string having the syntax:

```
{first,last}:pre:{person_id,user_id}:n
```

The 'first' or 'last' determines whether AME inserts the pre-approver before or after all other pre-approvers. The 'person_id' or 'user_id' indicates whether *n* (which must be a positive integer) is interpreted as a person or user ID. For example, the approval described as, "Make the approver with person ID 123 the first pre-approver." would correspond to the attribute value 'first:pre:person_id:123'.

Dynamic pre-approvals are useful when, for example, you need one functional analyst from a large group of analysts to pre-approve a certain class of transactions; but the particular analyst required varies from transaction to transaction, or when there are frequent changes in the identity of an analyst tasked with reviewing certain kinds of transactions. In such cases, it can be more convenient to compile a PL/SQL function that identifies the appropriate analyst, and have a dynamic pre-approver approval and its attribute select the attribute's value at run time, than to maintain an AME approval group for each analyst.

Dynamic Post-Approver (DEPRECIATED – Use Dynamic Groups)

The dynamic-post-approver approval type mimics the dynamic-pre-approver approval type, but for post-approvers. Replace 'pre' with 'post' in the syntax of the attribute value:

```
{first,last}:post:{person_id,user_id}:n
```

Maintaining Approvals

You maintain approvals and approval types using the Approvals tab.

You must have the Application Administrator responsibility to use the Approvals tab. If you are a business user, you view an appropriate list of approval types or approvals when you create or edit a rule. You cannot use the Approvals tab.

” **To view approval types and approvals:**

- Choose the Approvals tab. The approval-types list appears, listing every available approval type.

” **To create an approval type:**

See Appendix B for further details of how to create an approval type. Briefly, you:

1. Compile a PL/SQL *handler* (package or procedure) and make it executable by the APPS account.
2. Use the Approvals tab to create the approval type, which includes registering the handler.
3. Use the Approvals tab to create approvals for the new approval type.

Do not attempt to customize a seeded approval type's handler. Instead, copy the code, alter it as required, and register the result as a new approval type. This way you will avoid overwriting your handler customizations when you patch or upgrade your AME installation.

” **To add approvals to an approval type:**

1. Review the parameter syntax and semantics requirements given above for the appropriate approval type.
2. Choose the Approvals tab.
3. Select the name of the approval type to which you want to add approvals.
4. Choose the Add an Approval button.
5. Enter a parameter and description for the new approval on the Create an Approval page, and then choose the Create Approval button.
6. Repeat steps 3-4 until you have added all your approvals.

You can now choose the Quit button to return to the list of approval types.

Note: You never need to add approvals to the pre- and post-chain-of-authority approval types. When you create an approval group, the corresponding pre- and post-chain-of-authority approvals are automatically created.

” **To edit an approval type:**

1. Choose the Approvals tab.
2. Select the name of the approval type you want to edit.
3. Make your changes.
4. Choose the Submit Changes button.

You can now choose the Quit button to return to the list of approval types.

Note: You cannot edit seeded approval types.

” **To edit an approval:**

1. Choose the Approvals tab.
2. Select the name of the approval type whose approval you want to edit.
3. Select the description of the approval you want to edit.
4. Make your changes.
5. Choose the Update Approval button.

You can now choose the Quit button to return to the edit-an-approval-type page.

Note: You cannot edit seeded approvals, or pre- and post-chain-of-authority approvals. Pre- and post-chain-of-authority approvals are automatically updated when the corresponding approval group is updated.

” **To delete an approval type:**

1. Choose the Approvals tab.
2. Select the check box next to the approval type you want to delete, in the Delete column.
3. Choose the Delete Checked Approval Types button.
4. Confirm the deletion when prompted.

You can delete several approval types at once.

Note: You cannot delete seeded approval types.

” **To delete approvals from an approval type:**

1. Choose the Approvals tab.
2. Select the name of the approval type whose approval you want to delete.
3. Select the check box next to the approval you want to delete, in the Delete column.
4. Choose the Submit Changes button.
5. Confirm the deletion when prompted.

You can delete several approvals at once.

Note: You cannot delete a seeded approval, though you can delete an approval that has been added to a seeded approval type. Also, you cannot delete pre- and post-chain-of-authority approvals. Pre- and post-chain-of-authority approvals are automatically deleted when the corresponding approval group is deleted.

6
Approval Groups

Approval Groups

An approval group can either be an ordered set of one or more approvers (persons and/or user accounts) or it can be a list, which is dynamically generated at rule evaluation time. A typical pre- or post-approval rule adds an approval group's members (in order) to a transaction's approver list. Typically approval groups represent functional approvers outside a transaction's chain of authority, such as human-resource management and internal legal counsel, that must approve a transaction before or after management has done so.   However, it is possible to insert an approval group into a chain of authority if required.

When you create an approval group, AME creates for you the corresponding approvals of the pre- and post-approval approval types automatically. These approvals are available to all transaction types.

Approval groups can now be nested or in other works it is possible to make one approval group a member of another approval group. For example, a purchasing department might define the following groups to handle post-approvals of computer-hardware purchases:

COMP_APP_1 = {Jim Small}
COMP_APP_2 = {COM_APP_1, Jane Smith}
COMP_APP_3 = {COMP_APP_2, Liz Large}

AME would evaluate the membership of COMP_APP_3 to be (Jim Small, Jane Smith, Liz Large}, in that order.

You nest one approval group in another by going to the edit-group form on the groups tab and clicking the **Add Nested Group** button. The select list that appears on the **Choose a Nested Group** form only includes a group under three conditions:

- It is not the target group itself.
- It does not contain the target group (either explicitly or implicitly).
- It is not contained in the target group already.

The first two of these requirements prevent an infinite loop in the resolution of nested groups. The third prevents redundant group members only in the narrow sense that if group A already contains group B as a member, you cannot add group B to A again. However, if group A contains B, and group C also contains group B, you can add C to A. In this case, the ordering of A's members would place B's members before C's members other than those in B, in A's membership list. Thus:

B = {1, 2}

C = {3, 4, B}
A = {B, C} = {{1, 2}, {3, 4, B}} = {{1, 2}, {3, 4, {1, 2}}} = {1, 2, 3, 4}

Nested approval groups let you build an approvals matrix using AME approval groups. For example, the purchasing department defining the computer-hardware approval groups above might define three corresponding post-approval rules:

If ITEM_CATEGORY in {COMPUTER_HARDWARE}
and ITEM_AMOUNT <= 1,000 USD
then require post-approval from the COMP_APP_1 group.

If ITEM_CATEGORY in {COMPUTER_HARDWARE}
and 1,000 USD < ITEM_AMOUNT <= 10,000 USD
then require post-approval from the COMP _APP_2 group.

If ITEM_CATEGORY in {COMPUTER_HARDWARE}
and 10,000 USD < ITEM_AMOUNT
then require post-approval from the COMP_APP_3 group.

These rules effectively define a hierarchy of per-item dollar-amount signing authorities for three subject-matter approvers. You can seed a hierarchy of nested approval groups containing no members other than nested groups, along with a set of approval rules like those above; and your customers can easily populate the groups with person or user approvers upon installation.

It is possible to nest a dynamic group in a static group, but it is not possible to nest either a dynamic or static group in a dynamic group. Dynamic groups can only include persons and users.

AME maintains a non-recursive approver list for each approval group, so that the engine does not need to evaluate group nestings at run time. Rather, the engine fetches all static members and all dynamic members' queries in a single query, and then executes the dynamic members' queries in turn, substituting the results into the membership list. This means you can nest groups to arbitrary depth without impacting AME performance adversely, but you should take care to limit the number of dynamic groups nested in a single group.

Dynamic Approval Group

There are two new fields on the create-approval-group and edit-approval-group forms on the groups tab. One is a radio button labeled **Active List**; the other is a textarea labeled **Query**. The active-list radio button has two possible values: **static** and **dynamic**. When the radio button is set to **static**, AME uses the member list at the bottom of the (create or edit) form to determine the approval group's membership. When the radio button is set to **dynamic**, AME executes the query and uses the rows returned by the query to determine the approval group's membership.

The rules for the syntax and semantics of the query's results are simple. If the query is null, the group has no members when the radio button selects the value **dynamic**. (This lets you "switch off" a group without deleting its static members.) Otherwise, the query must select rows of the form **approver_type:id**, where *approver_type* is one of the strings 'person_id' and 'user_id', and *id* is a valid ID of the type specified by *approver_type*. For example, person_id:502 and user_id:205 are syntactically correct values. The query can reference the transaction-ID placeholder **:transactionId**. The query should select approvers in the order that you want them to appear in a transaction's approver list.

You can toggle the **Active List** button's values without deleting either the static list or the query string. The radio button's value merely determines which list the engine uses at run time.

There is a new required boolean attribute for the pre- and post-chain-of-authority approval types, **ALLOW_EMPTY_APPROVAL_GROUPS**. When this attribute has the value 'true', AME allows an approval group not to have any members. When the attribute has the value 'false', AME raises an exception if an approval group does not have any members.

Approval types other than the seeded pre- and post-chain-of-authority approval types can use approval groups. To fetch a group's membership, an approval type should always call ame_engine.getRuntimeGroupMembers. This procedure implements the functionality described in this FAQ. It returns approver-type values in the output parameter **parameterNamesOut**, and ID values in the output parameter **parametersOut**.

Dynamic approval groups supersedes the dynamic pre- and post-approver approval types which are now deprecated.

Maintaining Approval Groups

You view, create, edit and delete approval groups using the Groups tab.

" **To view approval groups:**

- Choose the Groups tab.

" **To create an approval group:**

1. Choose the Groups tab.
2. Choose the Add Group button.
3. On the Create an Approval Group page, enter a name and description that uniquely identify the new approval group.
4. For the Active List choose static if you are added members to the group or dynamic if you will be entering a selection using SQL.
5. Choose the Create Group button.
6. Options will be provided to add an approver, or add a nested group.
7. The approver-query wizard prompts you to search a group member which can be a person or fnd_user. The Nested Group will allow the addition of an approval group into the list. It is possible to add a mixture of groups and person/user to the list subject to the conditions mentioned above.
8. An order number is provided in both cases to enable the determination of the ordering of approvers in the group.
9. It is possible to change the ordering by selecting a group member and changing the order number on the update page.

You can now choose the Quit button to return to the list of approval groups.

Editing an approval group

There are several kinds of changes you can make to an approval group.

Note: You can edit the name and/or description of an approval group and delete group members in a single click of the Submit Changes button on the Edit an Approval Group page.

" **To change an approval group's name or description:**

1. Choose the Groups tab.
2. Select the name of the approval group that you want to edit.
3. Change the name and/or description.
4. Choose the Submit Changes button.

You can now choose the Quit button to return to the list of approval groups.

” **To add members to an approval group:**

1. Choose the Groups tab.
2. Select the name of the approval group to which you want to add a member.
3. Choose the Add Member button or Add Nested Group.
4. Either use the approver-query wizard to find the person or user account you want to add or select the required approval group.
5. Select the order number to insert the member/group at.
5. Repeat steps 3-5 until you have added all of the members/groups that you require.

You can now choose the Quit button to return to the list of approval groups.

” **To change the order of an approval group's members:**

1. Choose the Groups tab.
2. Select the name of an approval group.
3. Select the approver / group whose order you want to change.
4. Enter a new order number. The order number must be between 1 and the number of approvers /groups in the approval group.
5. Choose the Submit Changes button.

The group members now appear in the new order. You can now choose the Quit button to return to the list of approval groups.

” **To delete approval-group members:**

To delete one or more approval-group members:

1. Choose the Groups tab.
2. Select the name of an approval group.
3. Select the check box next to the approval group member's name, in the Delete column.
4. Choose the Submit Changes button.
5. Confirm the deletion when prompted.

You can now choose the Quit button to return to the list of approval groups.

” **To delete an approval group:**

To delete an approval group:

1. Choose the Groups tab.
2. Select the check box next to the name of the approval group you want to delete, in the Delete column.
3. Choose the Delete Checked Groups button.
4. Confirm the deletion when prompted.

You can delete several approval groups at once.

7
Rules

Approval Rules

Rule associate one or more conditions with an approval in an if-then statement. Before you can create rules, you must create conditions for the rules to use. You may need to create (or have a system administrator create) some custom attributes and/or approvals. You may also need to create some approval groups. Thus, while creating rules is your ultimate goal, it is also the *last* thing you do when you set up AME.

Rule Types

There are six *rule types* in AME:

- List-creation rules
- List-creation exceptions
- List-modification rules
- Substitutions
- Pre-list approval-group rules
- Post-list approval-group rules

Different rule types use different condition and approval types, and have different effects on a transaction's approver list.

Rule Priorities

The purpose of rule priorities is to prioritize a transaction type's rules and, at run time, remove from the set of rules that would otherwise apply to a transaction, those rules of insufficient priority. A *rule priority* is a positive integer associated with a rule within a transaction type. (Each transaction type that uses a rule can assign the rule a different priority.) **Note that priority increases as the priority value (number) decreases: two has priority over three, etc.** When rule priorities are enabled for a given rule type (within a given transaction type), the rules tab and the pop-up rules-details windows display rules' priorities; and one can edit a rule's priorities as one would edit any other rule property. When priorities are disabled, they are neither displayed nor editable.

The admin tab has a special form for editing the rulePriorityModes configuration variable's value. The value contains a priority mode and threshold for each rule type. There are three possible *priority modes*: absolute, relative, and disabled. A *threshold* is a positive integer, and it only has meaning for the first two priority-mode values. Full details of each of these modes is given in the chapter on AME Administration.

Rule Usages

The deployment of a rule depends upon its usage. It is possible to have multiple usages for a rule, each one spanning a specific data range which allows rules can be switched on/off as required.

Rule usage can also span transaction types, effectively sharing rules between them.

In both cases, any such usage is marked on the UI to indicate either future dated rules or rules shared across transaction type usages.

List-Creation Rules

AME uses *list-creation rules* to generate the default chain of authority for a given transaction. List-creation rules only use ordinary conditions (see Conditions: page 88 for an explanation of condition types). List-creation rules can use any of the following approval types:

- Absolute job level
- Relative Job Level
- Dual chains of authority
- Final approver only
- Manager then final approver, relative job level
- Supervisory level
- Approval group chain of authority
- Line-Item Job-Level chains of authority

See Approvals: chapter 5 for descriptions of each approval type.

This is an example of a list-creation rule:

Rule A

```
If
   TRANSACTION_AMOUNT < 1000 USD
then
   require approvals up to at least job level
2.
```

Rule A has one condition, on the attribute TRANSACTION_AMOUNT. The approval is of the absolute-job-level type. So if the condition is true for a given transaction, AME will extend the transaction's chain of authority from the transaction requestor's supervisor up to the first approver in the supervisory hierarchy who has a job level of at least 2.

If several list-creation rules of the same approval type apply to a transaction, AME enforces the most stringent approval required by those rules. For example, if two absolute-job-level rules apply to a transaction, the first requiring approvals up to at least job level 2 and the second requiring approvals up to at least job level 3, AME will extend the chain of authority up to the first approver with a job level of at least 3. So, when you want to create an exception to a list-creation rule, and the exception should require extra approval authority of the same type as the list-creation rule, the exception should itself be another list-creation rule.

List-Creation Exceptions

Sometimes you want to create an exception to a list-creation rule that decreases the level of, or changes the type of, the approval authority that the list-creation rule would otherwise require. In this case, you need a *list-creation exception* (or simply an *exception*). An exception contains at least one ordinary condition and at least one exception condition (see Conditions: page 88 for an explanation of condition types), as well as an approval. An exception can use any of the approval types available for list-creation rules.

The circumstances under which an exception overrides a list-creation rule are somewhat subtle. The two rules do not have to have the same ordinary conditions. Instead, both rules' ordinary conditions have to be defined on the same attributes, and both rules' conditions must all be true (including the exception's exception conditions). In this case, AME ignores the list-creation rule, and the exception has *suppressed* the list-creation rule.

There are several reasons AME does not require that an exception have the same ordinary conditions as a list-creation rule that it suppresses. First, one exception may be designed to suppress several list-creation rules. In this case, the scope of the exception's ordinary conditions must be broad enough to encompass that of each list-creation rule it suppresses. Second, an exception may be designed to apply to a more narrow set of cases than a list-creation rule. Third, it is sometimes desirable to adjust the scope of either the exception or the list-creation rule(s) that it suppresses, without simultaneously adjusting the scope of the other(s).

This is an example of an exception that suppresses Rule A:

Rule B

```
If
   TRANSACTION_AMOUNT < 500 USD and
   Exception: COST_CENTER is in {0743}
Then
   require approvals up to at least job level
1.
```

(In Rule B, the first condition is an ordinary condition, and the second condition is an exception condition.) Note that the ordinary condition is defined on the same attribute (TRANSACTION_AMOUNT) as the condition in Rule A, but the two conditions are different. Rule B carves out a exception to Rule A for transactions with totals under $500 U.S. for a certain cost center. In this narrow case, the exception requires less approval authority (of the absolute-job-level type) than what Rule A would otherwise require, which is why the rule must be an exception, rather than a list-creation rule.

List-Modification Rules

Sometimes you want to make exceptions regarding the approval authority granted to specific approvers, rather than the approval authority required for specific kinds of transactions. To do this, you need a list-modification rule. AME applies list-modification rules to modify the default chain of authority generated by all applicable list-creation and exception rules. A *list-modification rule* can have (but need not have) ordinary conditions. However, it must have exactly one list-modification condition (see Conditions: page 88 for an explanation of condition types).

There are two common uses for list-modification rules: reducing an approver's signing authority, and extending an approver's signing authority. In the former case, the list-creation rules might effectively grant a certain level of signing authority to managers having a given job level, and you might want not to grant that authority to a certain manager, in spite of their having the requisite job level. To do this, you tell AME to extend the chain of authority past the manager by using the nonfinal-authority approval type. In the latter case, just the opposite is true: the list-creation rules require approvals beyond a given job level, but you nevertheless want to grant a certain manager at that job level signing authority. To do this, you tell AME to truncate the chain of authority after the manager by using the final-authority approval type. In both cases, you can limit the scope of the list modification to a broad or narrow set of ordinary conditions.

Here are some examples of list-modification rules that illustrate the possibilities:

Rule C

```
If
   PURCHASE_TYPE is in {OFFICE FURNISHINGS,
OFFICE
   SUPPLIES} and
   Any approver is: Kathy Mawson
then
   Grant the approver final authority.
```

(In Rule C, the third condition is a list-modification condition.) Rule C grants Kathy Mawson final authority for a narrow scope of purchase types.

Rule D

```
If
   TRANSACTION_AMOUNT > 1000 USD and
   The final approver is: Kathy Mawson
then
   Require approvals at least one level up.
```

(In Rule D, the second condition is a list-modification condition.) Rule D revokes Kathy Mawson's signing authority for a broad set of transaction amounts.

Substitutions

Sometimes you want to delegate one approver's authority to another approver. To do this, you need a substitution rule. AME applies substitution rules to the chain of authority after modifying it by applying any applicable list-modification rules. Like a list-modification rule, a *substitution rule* can have (but need not have) ordinary conditions. However, it must have exactly one list-modification condition (see Conditions: page 88 for an explanation of condition types). So a substitution rule differs from a list-modification rule only in its use of the substitution approval type.

This is a sample substitution rule:

Rule E

```
If
   TRANSACTION_AMOUNT < 500 USD and
   CATEGORY in {MISCELLANEOUS OFFICE EXENSES}
and
   Any approver is: John Doe
```

```
then
   Substitute Jane Smith for the approver.
```

(In Rule E, the third condition is a list-modification condition.) Rule E delegates John Doe's authority to Jane Smith, for the class of transactions defined by the rule's ordinary conditions.

Pre- and Post-List Approval-Group Rules

The four rule types described above generate a transaction's chain of authority. You may want to have one or more groups of functional specialists approve a transaction before or after the chain of authority does. In such cases, you need a *pre-* or *post-list approval-group rule.* An approval-group rule must have at least one ordinary condition, and must use either the pre- or post-chain-of-authority-approvals approvals type. For example:

Rule F

```
If
   TRANSACTION_AMOUNT < 1000 USD and
   CATEGORY_NAME in {Marketing Event}
then
   Require pre-approval from Marketing
Approvals Group.
```

How AME Handles Multiple Requirements for an Approver

Sometimes an approval consisting of several rules of different types may require that a given approver appear in different places in a transaction's approver list. AME assumes that an approver should only approve a transaction once, so it has to decide which rule(s) requirements prevail. There are two common cases of this problem. Here are brief descriptions of each case, and explanations of how AME handles them:

1. A list-creation, exception, list-modification, or substitution rule includes the approver in the chain of authority; and a pre- or post-approval rule requires the approver's pre- or post-approval. In this case, AME only includes the approver in the chain of authority. The reason is that omitting the approver from the chain of authority might change the identity of the approvers that follow the approver, in the chain of authority; and AME assumes that preserving the chain of authority is more important than preserving the default approver order.
2. Two approval-group rules include the approver in different (pre or post) approval groups. In this case, AME includes the approver in the first approval group in the approver list.

AME here assumes that the approver should approve at the earliest opportunity.

How AME Sorts Rules at Run Time

At run time, AME decides the order in which to apply to a transaction all applicable approval rules according to the following algorithm:

1. Apply all exception rules first, to suppress any appropriate list-creation rules.
2. Apply all remaining list-creation and exception rules.
3. Apply any applicable list-modification rules.
4. Apply any applicable substitution rules.
5. Apply any applicable pre-approval rules.
6. Apply any applicable post-approval rules.

Within each step of the algorithm, AME sorts the rules by approval type, and processes all rules of a given approval type before processing any rules of another approval type.

Note: AME does not guarantee any particular ordering among the approval types, or among rules of a given type. For example, if several list-creation rules of different approval types apply to a single transaction, AME may process all of the rules of either approval type first. And if two substitution rules apply to a transaction, AME may process either rule first. Oracle encourages you to avoid relying on how your particular AME instance happens to handle an indeterminacy. Instead, try to craft your rules to avoid indeterminate outcomes.

You may nevertheless find it necessary to force AME to apply one rule before another. To do this, test the rules using the Test tab. If the orders are the reverse of what you want, edit the rules, swapping their contents without deleting the rules. If you follow this procedure, you should verify that AME has preserved the order of rule application each time you change your rules in any way, and each time you patch or upgrade AME.

Example Rule

Suppose you already have a rule requiring managerial approvals up to a manager with a job level of at least six for purchase requisitions totaling less than 5000 USD. You want to create an exception to this rule that requires approvals only up to a job level of at least four, when the requisition is for computer equipment. The rule you want would be of the exception type (not the list-creation type), because it *decreases* the level of approval authority required. So you would follow these steps to create the rule:

1. The pre-existing list-creation rule for the normal case would already have required a TRANSACTION_AMOUNT currency attribute, and a condition defined on that attribute. However, the Purchase Requisition transaction type might not include a suitable seeded attribute for a transaction's category. Supposing it does not, create a string attribute named (say) 'CATEGORY'. (If the attribute name already exists, share the attribute name. If not, create it with a sufficiently generic description for other transaction types to share the name, because the attribute name itself is sufficiently generic for several transaction types to want to use it.) Enter for the attribute a (dynamic) usage that selects a Purchase Requisition's category from the appropriate tables or views.
2. Create an *exception* condition on the CATEGORY attribute having only one allowed value, (say) 'COMPUTER EQUIPMENT'.

   You can now create the rule itself:
3. Enter the description 'computer equipment up to 5000 USD'.
4. Select the list-creation exception rule type.
5. Let the start date default to today.
6. Leave the end date blank, so the rule is in force indefinitely.
7. Select the absolute-job-level approval type.
8. Select the approval, 'Require approvals up to at least job level 4.'.
9. Select the ordinary-condition attribute TRANSACTION_AMOUNT.
10. Select the ordinary condition, '0 <= TRANSACTION_AMOUNT < 5000'.
11. Select the exception-condition attribute CATEGORY.
12. Select the exception condition, 'COMPUTER EQUIPMENT'.

Maintaining Rules

You can view, create, edit and delete rules using the Rules tab.

" **To display the list of rules for a transaction type:**

1. Choose the Rules tab.
2. Select the transaction type.

The transaction type's rules list appears. You add, edit, and delete rules from the rules list.

There are two versions of the rules list: short and long. The short list displays only the descriptions of the rules. The long list also displays the conditions, actions, and active attributes for each rule. The short list is displayed by default. To view the long list, choose the Display Long List button. (To return to the short list, choose the Display Short List button).

" **To create a rule:**

1. Display the list of rules.
2. Choose the Add a Rule button.
3. Enter a description for the rule. Make sure the description uniquely identifies the rule, and that it communicates the business case to which the rule applies.
4. Select a rule type. See Rule Types: page 129 for explanations of the rule types.
5. (Optional) Enter a start date. The start date defaults to today, but you can enter a future start date. You cannot enter a start date before today. Note that start dates always start at midnight; that is, a rule starts being in force at the beginning of its start date.
6. (Optional) Enter an end date. The end-date field is blank by default. If you leave the end date blank, the rule is in force indefinitely, once its start date arrives. End dates, like start dates, start at midnight; that is, a rule stops being in force at the beginning of its start date.
7. Select an approval type.
8. Select a specific approval of the type you selected in the previous step. For example, if you selected the absolute-job-level approval type, you must now select the absolute-job-level approval that reflects the job level you want the rule to require, and whether the rule should require at most or at least this job level.
9. If priority handling has been set, enter the priority for this rule. Please review the Administration chapter for further information.
10. Select the attributes used by the ordinary conditions that you want to include in the rule.

11. Select the ordinary conditions that you want to include in the rule.
12. If the rule is an exception, select the attributes used by the exception conditions that you want to include in the rule.
13. If the rule is an exception, select the exception conditions that you want to include in the rule.
14. If the rule is a list-modification or substitution rule, select the rule's list-modification condition.

You choose a Continue button at the bottom of each page in the wizard outlined above, to proceed to the next page. When you choose the Continue button at the end of the final step, AME saves the new rule and displays it on the rules list.

" **To edit a rule:**

1. Display the list of rules.
2. Select the description of the rule you want to edit.
3. Select the item you want to edit.
4. Edit the item, or select a replacement for it (depending on which item you selected in the previous step).

" **To delete a rule:**

1. Display the list of rules.
2. Select the check box next to the description of the rule you want to delete, in the Delete column.
3. Choose the Delete Checked Rules button.
4. Confirm the deletion when prompted.

You can delete several rules at once.

8
Testing

Testing Rules and Transactions

The Test tab has three features:

1. Fetch a transaction's attribute values.
2. View a transaction's approvals.
3. Create a test transaction.

These features enable you to view how AME processes real and fictitious transactions, without producing any notifications or otherwise interacting with another application.

How do I Test new Rules?

Here is the procedure Oracle recommends for adding one or more rules to your production AME instance:

1. Write a test plan that lists each business case your new rules should cover (and, ideally, which cases the new rules should *not* cover).
2. Create the new rules in a test environment, setting their start dates to today.
3. Use the Test tab to test the rules in the test environment. Refine the rules as needed, until they produce the outcomes you expect for all cases in your test plan.
4. Create the new rules in the production environment, setting their start dates *temporarily* to some future date.
5. Repeat the test in the production environment, using an effective date after the rule's temporary start date.
6. Change the rule's start dates to their actual start dates.

If you intend to add several rules to the production environment, always test them as a set in the test environment.

> **Warning:** ?It is possible to use the Test tab to test new or proposed rules in a production environment, without first testing the rules in a test environment. Oracle discourages this practice. If you do not have a separate test environment, or if for some other reason you must create a rule initially in your production environment, you should nevertheless follow the above procedure, skipping only steps 2 and 3. **If you create a rule with a start date of today before testing the rule, and the rule is for a transaction type whose application currently uses AME, the rule will apply to real transactions right away.**

Fetching a Transaction’s Attribute Values

You can display the attribute values for a selected transaction. The transaction’s header-level attribute values appear on one page, and the values of a given line-item-level attribute (for each line item) appear on a second page. This is useful for testing attribute usages you create or edit, and for troubleshooting at run time transactions whose workflows have stopped during their approval processes. It can also help explain the rule lists produced by the second Test-tab feature.

" **To view a transaction’s attribute values:**

1. Choose the Test tab.
2. Select a transaction type.
3. Select the Fetch a transaction’s attribute values radio button, then choose the Continue button.
4. Enter the ID of the transaction whose attribute values you want to view.
5. If you want to view a line-item attribute’s values, select the attribute’s name in the list of line-item attributes below the list of header-level attributes and their values.
6. Choose the Fetch Attribute Values button.

Viewing a Transaction's Approvals

You can display the approvals for a selected transaction. The rules that apply to a given transaction appear on one page, and the approver list generated by those rules appears on a second page. This is useful for testing rules by checking how they apply to real transactions, and for troubleshooting at run time transactions whose workflows have stopped during their approval processes.

" **To view a transaction's approvals:**

To view the list of rules that apply to a real transaction, and the resulting approver list:

1. Choose the Test tab.
2. Select a transaction type.
3. Select the View a transaction's approvals radio button, then choose the Continue button.
4. Enter the ID of the transaction whose approval process you want to view, then choose the View Approval Process button.
5. The list of applicable rules appears. To view the resulting approver list, choose the View Approver List button at the bottom of the rule list.

Creating a Test Transaction

You can create fictitious transactions and select the attribute values for the transactions. You can change a test transaction's attribute values as often as you like, to see how each change affects which rules apply to the transaction. In this way you can create test transactions representing distinct business cases, to make sure that each case invokes the rules you expect. It also lets you verify that a set of rules produces the approver lists you intend.

" **To create a test transaction:**

1. Choose the Test tab.
2. Select a transaction type.
3. Select the Create a test transaction radio button, then choose the Continue button.
4. A requestor-query wizard begins. Use it to search for the person or user who should be the transaction's requestor. (This query wizard gives you a user-friendly way to select a value for the mandatory attribute TRANSACTION_REQUESTOR_PERSON_ID or TRANSACTION_REQUESTOR_USER_ID.)
5. The Test Transaction page appears. Enter or select values for each mandatory attribute, and for each attribute that is active for the rules that you intend to apply to the test transaction. If you want to select or change a value for an attribute representing an ID (having a name ending with '_ID'), choose the Change an ID Attribute Value button at the bottom of the page. If the rules you want to test have active line-item attributes, choose the Edit Line Items button to create, edit, or delete test line items and values for the active line-item-attributes.
6. When you are satisfied with your test transaction's attribute values, choose the View Approval Process button at the bottom of the Test Transaction page to view the list of rules that apply to your test transaction, and the approver list they generate.
7. To modify and resubmit the test transaction, choose the Change Attribute Values button at the bottom of the Test-Transaction Results page. This returns you to step 5.

9
Administration

Administration

The Admin tab is available only to users with the Application Administrator responsibility. You can use the Admin tab's features to maintain AME's configuration variables and transaction types, and to analyze AME's runtime exceptions.

Configuration Variables

AME defines a number of configuration variables. In all cases, the configuration variables have default values that are created when AME is installed. In some cases, each transaction type can override the default value with its own value as well. Configuration-variable names and values are case-sensitive. Here are brief explanations of the configuration variables, indicating in each case whether a transaction type can override the default value.

adminApprover

The adminApprover variable identifies the person or user account that AME identifies as a transaction's next required approver to the application requesting the approver's identity, when AME encounters an exception while generating the transaction's approver list. A transaction type can override this variable's default value.

A widget is provided to select the person or the user who is the adminApprover.

currencyConversionWindow

The currencyConversionWindow variable identifies how many days AME should look back , at most, to find the a currency conversion rate. The default value is set to 120 days. AME uses the GL Daily Rates table and associated routines to perform currency conversions. [complete]

distributedEnvironment

The distributedEnvironment variable indicates whether AME has been installed in a distributed-database environment. It has two possible values: 'yes' and 'no'. A transaction type cannot override this variable's default value.

AME has its own exception log, and in most cases it logs runtime transactions to that log. If the application that owns a given transaction type calls AME from within a workflow, AME also logs exceptions to Workflow (see Runtime Exceptions: page 116 for details); and it uses an autonomous transaction to commit exception data to its own log, to avoid interfering with Workflow's transaction management.

If however AME is installed in a distributed-database environment, and is called from within a workflow, it cannot initiate an autonomous transaction, because such transactions are not yet possible in a distributed environment. In this case it must query Workflow directly (where possible) for a record of AME exceptions. This log is less robust than AME's own log, and so AME avoids using it where possible.

In short, the distributedEnvironment variable is necessary to make sure AME logs exceptions internally whenever possible, while avoiding autonomous transactions where they would produce runtime errors.

forwardingBehaviors

The forwardingBehaviors screen defines a set of constants which determines how AME handles forwarding in a number of special cases.

The behavior for forwarding to someone not already in the list is always the same: the forwardee is inserted as an approver of the same type, immediately after the forwarder. When the forwarder and forwardee are chain-of-authority approvers, and the forwarder lacks final authority, AME extends the chain of authority starting from the forwarder until it finds someone with final authority. (This would normally be the typical case.).

AME will seeds default values that are expected to be the normal usage for each forward type. Oracle Application teams seeding transaction types can override these defaults to ones more suitable for the particular business process.

There are a number of different types of forwarding scenario that might occur during the approval process. For each of the listed scenario below, the AME engine can be instructed to amend the approver list in a pre-defined way. Not all possible outcomes are applicable for each forwarding scenario, these all indicated below.

The following list details the possible options, the outcome of which define how the approver list is amended.

Remand

All approvers starting with the forwardee up to but not including the forwarder are added to the approval list.

Forward to forwardee and forwarder

The forwardee and then the forwarder are inserted into the approver list after the forwarder.

Forward to forwardee only

The forwardee is inserted into the approver list after the forwarder.

Ignore forwarding

The forwarding is not executed. If the forwarder forwards without approval the chain of authority will be extended past the forwarder to the next person in the hierarchy.

Repeat Forwarder

The forwarder is included again in the same chain of authority.

Skip Forwarder

The forwarder is skipped when the chain of authority is extended from the forwardee.

There are two specific types of forwarder handled. The forwarder either exists in a chain of authority or is an ad hoc approver. Each scenario is explained along with a list of applicable forwarding actions.

Chain of Authority Forwarder

Previous approver, same chain of authority

The forwarder forwards approval to a previous approver that exists in the same chain of authority. This may typically be the case where the approver is questioning the approval. The forwarding may have occurred in one of the two cases

The forwarder forwards without approval
The allowable outcomes are: Remand, Forward to forwardee and forwarder (default), Forward to forwardee only, Ignore forwarding

The forwarder forwards with approval
The allowable outcomes are: Remand, Forward to forwardee and forwarder, Forward to forwardee only (default), Ignore forwarding

Subordinate not in same chain but in same hierarchy

The subordinate does not exist in the chain of authority. This situation may exist because the approval chain originally started above the subordinate. In this circumstance it may be necessary, depending upon the desired outcome, to ascend the hierarchy starting at the subordinate and requiring approvals up to but not including the forwarder. If the desired outcome is Skip Forwarder, the supervisor of the forwarder is added to the approver list. If the outcome is Repeat Forwarder, the forwarder is required to approve again.

The forwarder forwards without approval
The allowable outcomes are: Forward to forwardee and forwarder, Forward to forwardee only, Repeat forwarder (default), Skip forwarder, Ignore forwarding

The forwarder forwards with approval
The allowable outcomes are: Forward to forwardee and forwarder, Forward to forwardee only (default), Repeat forwarder, Skip forwarder, Ignore forwarding

Already in list but not in same hierarchy

The forwardee appears in the approver list but is not within the same hierarchy as the forwarder. This may be the case if the forwardee is a pre-approver or is included within a different chain of authority or is included within a group that has been inserted into the chain of authority.

The forwarder forwards without approval

The allowable outcomes are: Remand, Forward to forwardee and forwarder, Forward to forwardee only (default), Ignore forwarding

The forwarder forwards with approval

The allowable outcomes are: Remand, Forward to forwardee and forwarder, Forward to forwardee only (default), Ignore forwarding

Ad-Hoc Forwarder

In this instance the Forwarder is an Ad-hoc approver that appears either in Pre, Post or is an ad-hoc authority approver. In these cases the selectable outcomes are constrained because only the forwarder and forwardee are the actors when the forwarding takes place, i.e. there is no hierarchal chain to ascend within the context of the two approvers.

The selectable outcomes are only valid when the forwardee is already in the approver list.

The forwarder forwards without approval

The allowable outcomes are: Forward to forwardee and forwarder, Forward to forwardee only (default), Ignore forwarding

The forwarder forwards with approval

The allowable outcomes are: Forward to forwardee and forwarder, Forward to forwardee only (default), Ignore forwarding

helpPath

The helpPath variable identifies the absolute virtual path to AME's help files. It must conform to the syntax,

```
http://server[:port]/virtual_path/
```

(port numbers are only required for ports other than the default port 80). For example:

```
http://myAppsServer/helpFiles/
```

is a syntactically valid helpPath value. A transaction type cannot override this variable's default value.

htmlPath

The htmlPath variable identifies the relative virtual path to AME's HTML files other than its help files. The value must start and end with forward slashes. For example:

```
/OA_HTML/
```

is a syntactically valid htmlPath value. A transaction type cannot override this variable's default value.

imagePath

The imagePath variable identifies the relative virtual path to AME's image files. The value must start and end with forward slashes. For example:

```
/OA_MEDIA/
```

is a syntactically valid imagePath value. A transaction type cannot override this variable's default value.

portalUrl

The portalUrl variable identifies the absolute URL to which AME's portal icon is hyperlinked. It must adhere to HTTP's syntactic requirements for absolute URLs. A transaction type cannot override this variable's default value.

purgeFrequency

The purgeFrequency variable indicates how many days AME should preserve temporary data before purging it from AME's database tables. The value must be a positive integer.

When a transaction's temporary data is purged, its approval process starts over, as if no approver had approved the transaction. Therefore, the purgeFrequency variable's value should be high enough so that no transaction will require this many days to be approved by all of its approvers. At the same time, the purge frequency should be sufficiently low to avoid unnecessary growth in the size of AME's temporary-data tables.

A transaction type can override this variable's default value. When a transaction type overrides purgeFrequency, AME preserves that transaction type's temporary data only for the overriding value. This enables you to set the purge frequency differently for each transaction type, adjusting its value for each to a reasonable upper limit on the number of days required for a transaction of that type to complete its approval process.

repeatedApprovers

Indicates how many times to require an approver's approval in the absence of forwarding. An approver may appear many times within the overall approval list. This can be due to a number of factors such as the approver exists as a pre/post approver as well as appearing within a chain of authority. In these circumstances it may be desirable to restrict so that the approver is only required to approve once in the following circumstances.

One of the following three options should be selected.

- Once per transaction
- Once per sublist
- Once per group or chain

A sublist can be either pre approver, chain of authority, or post approver.

rulePriorityModes

The rulePriorityModes defines for each rule type either the threshold for absolute or priority processing or it disables priority processing altogether for the specific rule type.

For each rule type select one of the following

Disabled
: Priority processing is disabled for the rule type

Absolute
: Used to exclude rules that have a rule priority value numerically greater than that of the threshold. This mode can be used to remove rules from temporary use.

Relative
Used to preserve the top 'n' rules where 'n' is the threshold value. For example if AME were to determine that 5 rules were satisfied by a transaction and the threshold was set to 3 then the first three rules, in order of priority, would be include and the rest discarded.

Threshold
: The threshold value for either Absolute or Relative priority rule processing. The threshold value is a positive integer and defines the point where rules are included in rule processing. The effect of this threshold

is described above in the settings for Absolute and Relative processing.

useWorkflow

The useWorkflow variable indicates whether the application that owns a given transaction type calls AME from within a workflow. It has two possible values: 'yes' and 'no'. AME uses this variable's value in conjunction with that of the distributedEnvironment variable (see distributedEnvironment: page 112) to determine how to log runtime exceptions. Every transaction type *should* override this variable's default value.

Transaction Types

In AME, a *transaction type* represents a set of transactions generated by a given application, for which AME maintains a single set of approval rules. A single application can have several transaction types. For example, the Web Expenses self-service application represents one of several possible transaction types for Accounts Payable.

How AME Identifies Transaction Types

Transaction-Type Descriptions

AME displays a user-friendly description for each transaction type. This description appears throughout AME's Web interface in pop-down lists from which users select the transaction type they want to work in on a given tab. You can edit this description using the Admin tab's Maintain transaction types radio button.

Application and Transaction-Type IDs

AME's API (by which applications communicate with AME) identifies a transaction type by the combination of the fnd_application.application_id value of the application that owns the transaction type, and a transaction-type identifier (a string up to 50 characters long). If the application only has one transaction type, the transaction-type identifier may be null. Otherwise, its value is typically (though not necessarily) the appropriate Workflow item type for the transaction type, and it must be unique among the transaction-type IDs of the transaction types owned by the application.

Internally, AME assigns a separate ID to each transaction type. You use this ID as the value of the ame_internal_trans_type_id secured attribute, in conjunction with the Limited Business responsibility, to give a user business access to a transaction type.

A custom application may use AME to manage its approval process. Such an application would presumably not have an fnd_application.application_id value, so it would have to pass an unassigned integer (such as a negative integer) to AME's API. See The AME API: page - 160 for details.

Other Transaction-Type Data

Attribute Usages and Configuration-Variable Values

Transaction types also define attribute usages, and can override certain configuration variables. See Attributes: page 75 and Configuration Variables: page 112 for details.

Line-Item-ID Query

A transaction type may (but is not required to) define a line-item-ID query. This is necessary for the transaction type to define usages for line-item attributes (and so to define conditions and rules that reference line-item attributes). The line-item-ID query must select a single number column, and it must order the results in ascending order. AME will raise an exception at run time if a line-item-ID query fails to order the line-item IDs in ascending order, and it will not let you enter a line-item-ID query that does not include an order-by clause.

Runtime Exceptions

What Causes Runtime Exceptions in AME?

The most common reason AME raises an exception (which typically results in the related application's stopping a transaction's workflow) is that AME cannot ascend a hierarchy, either because a slot in the hierarchy is vacant, or because an approver's level in the hierarchy is indeterminate. For example, in the case of the HRMS supervisory hierarchy, an approver may have a null supervisor or a null job level. In this case, the missing data must be entered into the appropriate application before restarting the offending transaction's workflow.

What happens when AME raises an exception?

When AME cannot determine a transaction's next required approver (in response to a request from an application, or when you use the Test tab), it:

1. raises an exception in the routine that has trouble generating the approver list, and re-raises the exception up its call stack until an AME API routine catches the exception. Note that AME does *not* generally raise the exception to the routine that called its API.

2. logs each exception to its internal exception log (where possible), and to Workflow (when the AME API was called from a workflow in another application).
3. (if AME was responding to a request from another application) identifies as the next required approver the person or user account identified by the appropriate value of the adminApprover configuration variable, and sets the approval status of this approver to ame_util.exceptionStatus. (This is the only circumstance where AME uses this approval-status value.)

The requesting application may or may not notice that AME has identified an administrative approver as the next required approver, or that AME has set the approver's status to indicate that an exception has occurred. If it does, it typically will respond by stopping the transaction's workflow and notifying a Workflow system administrator. In this case, the person or user identified by the adminApprover configuration variable will not at this time receive a notification regarding this transaction (unless that person happens to be the Workflow system administrator as well). The application may elect instead merely to send a notification to the administrative approver identified by AME, indicating that an exception has occurred for the transaction.

If the requesting application does not notice that the next required approver is an administrative approver, it will treat the administrative approver as it would any other approver: by sending the approver a notification requesting their approval of the transaction. The approver would then have to discern that AME had encountered an exception while attempting to calculate the transaction's approver list.

Oracle recommends that you configure the adminApprover configuration variable to identify the same individual as the Workflow and AME administrator for a given transaction type. This will have the effect of making sure the same individual is always notified when that transaction type's workflow errors, regardless of whether the error arises within AME.

How Should an Administrator Respond to an AME Exception?

However a Workflow system administrator or AME administrative approver becomes aware that AME is having trouble processing a transaction, they should respond to the problem as follows:

1. Check AME's exception log for the transaction (see View a Transaction's Exception Log: page 120 for details).
2. Check Workflow's context log for any other relevant details.
3. Correct the (usually data) problem that caused AME difficulty.
4. Restart the transaction's workflow.

5. Clear the transaction’s exception log in AME (see Clear a Transaction’s Exception Log: page 122 for details).

Updating Configuration-Variable Values

The Admin tab is available only to users with the Application Administrator responsibility.

" **To update default configuration-variable values:**

The Admin tab's first radio button enables you to set default values for AME's configuration variables. To edit a configuration variable's default value:

1. Choose the Admin tab.
2. The Update default configuration-variable values radio button is selected by default. Choose the Continue button.
3. Select the name or description of the variable whose value you want to change.
4. Edit the value.
5. Choose the Submit Changes button.

" **To update transaction-type configuration-variable values:**

The Admin tab's second radio button enables you to set values for certain configuration variables, for a given transaction type. The values you set here will override the default values, for the transaction type. To edit a transaction type's configuration-variable values:

1. Choose the Admin tab.
2. Select the Update transaction-type configuration-variable values radio button, and choose the Continue button.
3. Select a transaction type.
4. The Edit Transaction Type Configuration Variables page appears. It asks you several questions whose answers determine the values of the useWorkflow, purgeFrequency, adminApprover, etc. configuration variables. Edit the answers to those questions to your satisfaction, and choose the Continue button. (AME saves the changes related to the useWorkflow and purgeFrequency variables when you submit this page.)
5. If you selected "a new business owner" as your answer to the final question, the business-owner-query wizard starts. Use the wizard to search for the person or account that you want to be the administrative approver for the transaction type. (AME saves the new adminApprover value when you complete this wizard.)

Maintaining Transaction Types

The Admin tab's third radio button enables you to edit existing transaction types and add new ones. All transaction types must be registered with AME, before they will appear on AME's pop-down transaction-type lists. Generally you do not need to use this radio button to register new transaction types; Oracle Application patches and upgrades will generally do it for you. You may want to register a transaction type for a custom or third-party application. In this case, you will need to use this radio button.

" **To register a transaction type:**

1. Choose the Admin tab.
2. Select the Maintain transaction types radio button, and choose the Continue button.
3. Choose the Add Transaction Type button.
4. The Choose an Application page appears. Select an application to own the transaction type.
5. The Enter Registration Details wizard begins. Enter a user-friendly description for the transaction type. Enter a transaction-type ID if required (see Application and Transaction-Type IDs: page 115 for details). If you want to enable line-item attributes for the transaction type, enter a line-item-ID query (see Line-Item-ID Query: page 116 for details).
6. The Mandatory Attribute Query Entry page appears. Select usage types and enter usages for each mandatory attribute, for the new transaction type.
7. The Transaction Type Configuration Variables page appears. This is the same functionality that the Update transaction-type configuration-variable values radio button uses. See Update Transaction-Type Configuration-Variable Values: page 118 for details about how to use this functionality.

" **To edit a transaction type:**

To edit a transaction type's configuration-variable values, use the Update transaction-type configuration-variable values radio button. See Update Transaction-Type Configuration-Variable Values: page 118 for details. To edit a transaction type's mandatory-attribute usages, use the Admin tab. See Maintaining Attributes: page 84 for details.

To edit a transaction type's description, transaction-type ID, or line-item-ID query:

1. Choose the Admin tab.
2. Select the Maintain transaction types radio button, and choose the Continue button.
3. Select the description of the transaction type you want to edit.
4. Check an item(s).
5. Choose the Submit Changes button.

" **To delete a transaction type:**

To delete a transaction type (including all of its attribute usages and rules):

1. Choose the Admin tab.
2. Select the Maintain transaction types radio button, and choose the Continue button.
3. Select the check box next to the transaction type you want to delete, in the Delete column.
4. Choose the Delete Checked Items button.
5. Confirm the deletion when prompted.

   Note: When you delete a transaction type, any rules that it shares with other transaction types are not deleted (for those transaction types).

" **To view all exceptions for a transaction type:**

You can check whether a particular transaction type is regularly encountering a certain exception, or sequence of exceptions, by viewing the transaction type's exception log. The log displays all uncleared exceptions for the transaction type. To view the log:

1. Choose the Admin tab.
2. Select the View all exceptions for a transaction type radio button, and choose the Continue button.
3. Select a transaction type.
4. The transaction type's exception log appears, with the exceptions sorted in descending log-ID order. (This order indicates the order in which the exceptions were logged.) If you want to sort the exceptions by the names of the PL/SQL package and routine that raised the exceptions, choose the Sort by Package, Routine button.

" **To view a transaction's exception log:**

A transaction's exception log can help you identify the data problem(s) that led AME to raise an exception, which typically stops the transaction's workflow in the application originating the transaction. (See Runtime Exceptions: page 116 for details.) To view a transaction's exception log:

1. Choose the Admin tab.
2. Select the View a transaction's exception log radio button, and choose the Continue button.
3. Select a transaction type.
4. Enter the transaction ID.
5. The transaction's exception log appears, with the exceptions sorted in descending log-ID order. (This order indicates the order in which the exceptions were logged.) If you want to sort the exceptions by the names of the PL/SQL package and routine that raised the exceptions, choose the Sort by Package, Routine button.

" **To view the Web interface's exception log:**

The Web interface should not raise exceptions during normal AME operation. You may find it necessary to view the exception log for AME's Web interface (most likely while working with Oracle Support). To view the log:

1. Choose the Admin tab.
2. Select the View the Web interface's exception log radio button, and choose the Continue button.
3. The Web interface's exception log appears, with the exceptions sorted in descending log-ID order. (This order indicates the order in which the exceptions were logged.) If you want to sort the exceptions by the names of the PL/SQL package and routine that raised the exceptions, choose the Sort by Package, Routine button.

" **To clear a transaction type's exception log:**

In the unlikely event that a transaction type develops a problem not due to data errors, you may need to clear the transaction type's entire exception log. Make sure that before you do so, you solve the underlying problem and restart the workflows of all transactions that appear in the exception log. Clearing an exception log is irreversible.

To clear a transaction type's exception log:

1. Choose the Admin tab.
2. Select the Clear a transaction type's exception log radio button, and choose the Continue button.
3. Select a transaction type.
4. Confirm the action when prompted.

" **To clear a transaction's exception log:**

You should clear a transaction's exception log only after solving the (typically data) problem that gave rise to the exception. (See Runtime Exceptions: page 116 for details.)

To clear a transaction's exception log:

1. Choose the Admin tab.
2. Select the Clear a transaction's exception log radio button, and choose the Continue button.
3. Select a transaction type.
4. Enter the transaction's ID.
5. Confirm the action when prompted.

" **To clear the Web interface's exception log:**

In the unlikely event that the Web interface raises one or more exceptions, you should clear the Web interface's exception log only after solving the problem that occasioned the exceptions, probably with the help of Oracle Support. To clear the log:

1. Choose the Admin tab.
2. Select the Clear the Web interface’s exception log radio button, and choose the Continue button.
3. Confirm the action when prompted.

Appendix A
How AME Processes its Rules at Run Time

How AME Processes Rules at Run Time

An application that uses AME to manage its transactions' approval processes communicates with AME through an extensive API (see The AME API: page 163 for details). Typically , the application follows these steps, for a given transaction:

1. Request the identity of the transaction's next approver from AME.
2. If there is no next approver, the transaction's approver process is complete.
3. Otherwise, request approval from the next approver.
4. When the approver responds to the request for approval, pass the response to AME (which stores the response), and go to step 1.

Each time an application takes step 1 in the algorithm above, AME recalculates the transaction's approver list, to make sure the approver list represents the current transaction attribute values, the current approval rules, and the current organization structure. The recalculation also accounts for various kinds of responses from the most recent approver; in particular, forwardings (with or without approval). AME follows these steps, to (re)calculate a transaction's approver list:

1. Fetch the values of the relevant transaction type's active attributes.
2. Evaluate the conditions in each active rule in the transaction type's set of rules to see which rules apply to the transaction. (Recall that a rule is active if its start date precedes the value of the EFFECTIVE_RULE_DATE mandatory attribute; and if either the rule's end date is null, or the end date follows the value of EFFECTIVE_RULE_DATE. Also, recall that a rule applies to a transaction if all of the rule's conditions are true, and that a condition is true if the value of the attribute on which the condition is defined lies in the condition's set of allowed values.)
3. Sort the rules by rule type.
4. Apply the rules one type at a time.
5. Insert into the approver list any approvers that the application owning the transaction has submitted to AME for insertion. (AME keeps a record of such dynamic approver insertions between calls to its API, so the inserted approvers appear in the approver list each time AME calculates the list. For details about AME's dynamic-approver-insertion capabilities, see The AME API: page 163).
6. Delete from the approver list any approvers that the application owning the transaction has submitted to AME for deletion, if the mandatory boolean attribute:

   ```
   ALLOW_DELETING_RULE_GENERATED_APPROVERS
   ```

is true. (AME keeps a record of such dynamic approver deletions between calls to its API, so the deleted approvers are removed from the approver list each time AME calculates the list. For details about AME's dynamic-approver-deletion capabilities, see The AME API: page 163.)

7. Compare the current approver list with AME's approver-status history to identify the first approver on the current list that has not approved the transaction. If such an approver exists, return this approver's identity to the requesting application. Otherwise, return null to the requesting application to indicate that the transaction's approval process is complete.

The fourth step of the above algorithm itself represents another algorithm. How AME Sorts Rules at Run Time: page 134 summarizes how AME sorts rules by type and applies them one type at a time, in the following order:

1. exceptions and list-creation rules not suppressed by the exceptions
2. list-modification rules
3. substitution rules
4. pre-approval rules
5. post-approval rules.

Again, the first step represents yet another algorithm:

1. Sort the (list-creation and exception) rules by their approval types.
2. For each approval type:

   A. If the value of the mandatory boolean attribute:

   ```
   ALLOW_REQUESTOR_APPROVAL
   ```

   is true, have the approval type's handler check whether the requestor has signing authority for this transaction. If so, continue to the next approval type.

   B. Get the identity of the next approver from the approval type's handler, and add that approver to the end of the approver list.

   C. If the latest approver has already responded to a request for approval of the current transaction by forwarding the request to another approver, add the forwardee to the end of the approver list.

   D. Have the approval type's handler check whether the latest approver has signing authority for this transaction. If so, continue to the next approval type. Otherwise, go to step B.

Each time AME communicates with an approval type's handler, it passes the handler the set of approval parameters of the rules of the current approval type that apply to the transaction being processed. The handler uses the parameters to decide where to start and end the chain of authority for the current approval type.

AME processes list-modification, substitution, pre-approval, and post-approval rules in much the same way that is processes list-creation and exception rules. In each case, AME sorts the rules by approval type, and processes all rules of a given approval type together. The appropriate approval type's handler performs each of the approvals required by the rules using that type.

Appendix B
How To Create A Custom Approval Type

How to Create a Custom Approval Type

There are four steps involved in creating a custom approval type:

1. Code a handler PL/SQL package or procedure for the approval type, and give the APPS account execute privileges on it.
2. Create the approval type using the Approvals tab.
3. Create approvals for the new approval type.
4. Test the approval type.

This appendix contains detailed instructions for the first two steps. See Adding Approvals to an Approval Type: page 103 for instructions regarding step 3.

What do I Need to Know Before I Start?

Categories of Approval Types

There are three different kinds of approval-type handlers: list-creation (authority) handlers, list-modification handlers, and approval-group handlers. AME has six rule types. Each type can only use a certain category of approval type:

Table 1: Categories of Approval Types

Rule Type	Category of Approval Type
list creation (authority)	list creation (authority)
Exception	list creation (authority)
list modification	list modification
Substitution	list modification
pre-authority-list approval group	approval group
post-authority-list approval group	approval group

Every approval type is implemented as a PL/SQL procedure or package *handler* (depending on the approval type's category). The rest of this section specifies the syntax and functionality requirements that AME imposes on each category of approval-type handler.

When to Write an Approval-Type Handler

You should create a custom list-creation handler if none of the standard AME approval types (and their handlers) represents the approval-authority hierarchy that your organization wants to implement. (This is the most likely reason to write a custom handler.) You should create a custom list-modification handler if none of the standard AME list-modification handlers lets you modify authority-based approval lists according to your organization's business rules. Finally, you should write a custom approval-group handler if your organization wants to include nonstandard approval-group functionality in its approval processes—for example, selecting a random subset (of fixed size) of an approval group to pre- or post-approve a transaction, or implementing approval groups with hierarchical structure.

List-Creation-Handler Efficiency

AME's engine makes a number of calls to a list-creation handler for each transaction requiring the corresponding approval type. If the number of such transactions processed by AME daily is at all significant, you should be very careful to design and code your handler with efficiency as your foremost concern. This document describes transaction- and handler-state-maintenance mechanisms that a handler can sometimes use to improve efficiency by up to an order of magnitude. Make sure you understand whether, why, and how to maintain state in your handler code for the sake of efficiency.

Caution

AME's fundamental architectural principle is to encode in approval rules all decisions about the general structure of a transaction's approval process. When you write an AME approval-type handler, you have the ability to violate that principle by coding such decisions into the handler. You do your organization a disservice when you "hard code" such decisions, because they then become invisible to the business people whose responsibility is to define the business rules that determine transactions' approval processes.

> **Attention:** ?Please make sure your handlers only translate AME rules' general structural requirements into specific person and/or user IDs.

Referencing Engine Package Variables

Handler code may have occasion to reference any of the following ame_engine package (global) variables:

Table 2: Engine Package Variables

Variable	Data Type	Description
tempAmeApplicationId	integer	internal unique identifier for a transaction type; stored in ame_calling_apps.application_id
tempFndApplicationId	integer	fnd_application.application_id value of application that owns the current transaction; value of applicationIdIn input argument in ame_api routines
tempTransactionId	varchar2(50)	unique identifier for a transaction; value of transactionIdIn input argument in

		ame_api routines
tempTransactionTypeId	varchar2(50)	possibly null transaction-type identifier; value of transactionTypeIn input argument in ame_api routines; combination of an FND application ID and a transaction-type ID corresponds to a unique AME application ID in ame_calling_apps
tempApproverList	ame_util.approversTable	approver list for current transaction; modified directly by list-modification and approval-group handlers, but not by authority handlers; kept compact

The global variables are public (declared in the ame_engine package header) to allow handlers and other runtime code to access the variables directly, which avoids the overhead that would otherwise be associated with accessing the variables *via* wrapper routines. This means you must exercise caution in referencing the global variables.

The values of the first four variables in Table 2 are set before the engine calls a handler. A handler should only read them, and never modify them.

Warning: ?Modifying these values will result in engine exceptions at run time.

The engine builds a transaction's approver list in tempApproverList. Authority handlers must not modify the variable directly (the engine does that for them), though they may read it to determine who is currently in the list. List-modification and approval-group handlers must modify the variable directly. It is critical that tempApproverList be kept compact, meaning that its indexes start at one and ascend the positive integers without skipping any. So, if a handler needs to delete an approver in the list, it must call ame_engine.compactApproverList immediately after the deletion, to compact the list.

Warning: ?Failure to compact tempApproverList after deleting an approver will result in engine exceptions at run time.

Fetching Required Attributes' Values at Run Time

A handler procedure may need to know the values of certain transaction-specific decision variables, to make the decisions that this document requires of them. For example, the hypothetical ame_military_rank_handler.getFirstApprover procedure discussed under Authority Handlers: page 147 might need to use the person ID of a transaction's requestor to fetch the user ID of the requestor's commanding officer, in order to return the commanding officer's user ID.

Whenever practical, such decision variables should be represented within AME as attributes. Then, when you create the approval type, AME will prompt you to select the attributes that your new approval type requires. Thereafter AME will require all applications using your approval type to first provide AME with usages for these attributes, before the applications can use your approval type in their rules. This approach makes it possible for several transaction types to share your handler safely.

To fetch an attribute's value, a handler must call either the ame_engine function getAttributeValueByName (for single-valued attribute types) or the ame_engine procedure getAttributeValuesByName (for currency attributes). Attribute names (in the attributeNameIn argument) are always in upper case.

Identifying Approvers

There are two ways to identify an approver in AME: as a user account (an fnd_user.user_id value) or as a person (a per_all_people_f.person_id value). Every input or output argument to a handler procedure that identifies an approver is of the (possibly serialized) type ame_util.approverRecord, which has user_id and person_id fields. Your handler procedures must make sure that at least one of these fields is non-null in all of the output arguments.

If your handler can only process one type of ID, and it receives an input argument containing only the other type of ID, your handler should call one of the ame_engine conversion routines getPersonId and getUserId (as appropriate) to convert the ID to the other type. The getPersonId function returns the (possibly null) fnd_user.employee_id value corresponding to the user_id *userIdIn*, or null if it finds no rows with user_id value *userIdIn*. The getUserIds procedure fetches the fnd_user.user_id values of all rows having employee_id value *personIdIn*, and returns the user_id values in *userIdsOut*. If getUserIds finds no matching rows, it returns ame_util.emptyIdList (which is just an empty PL/SQL table). If your code calls getUserIds, it should use the user_id in the first row of *userIdsOut*, unless your code includes logic to sort through the user_ids in *userIdsOut* and determine which of them to use, according to rules specific to your approval type.

If your handler requires one type of ID, it receives only the other type of ID as an input argument, and the appropriate ame_engine conversion routine fails to convert the ID to the other type, your code should raise an exception according to the instructions under Raising Exceptions: page 145.

(De)serializing Handler Arguments

The AME engine uses dynamic PL/SQL to execute your handler routines at run time. Dynamic PL/SQL does not permit the use of compound data types (records or tables of records) as bind variables for a procedure's arguments. So AME's ame_util package includes conversion routines that *serialize* (convert from another data type to a string) and *deserialize* (convert the string back to the data type) various ame_util data types used by the engine. Where the AME engine needs to pass a compound data type to a handler routine, or receive one from a handler routine, this document's procedure specifications identify the serialized arguments. Always use the appropriate ame_util (de)serialization routines to (de)serialize these arguments' values.

The ame_util serialization routines use the character constant ame_util.fieldDelimiter (currently a comma) as a field delimiter within records, and the character constant ame_util.RecordDelimiter (currently a semicolon) as a record delimiter within tables. A parameter list's native data type is an ame_util.parameterList, which is a table of varchar2; so a serialized parameter list uses ame_util.fieldDelimiter as a field delimiter. Therefore, your approvals' parameters must not contain ame_util.fieldDelimiter. The AME user interface will display an error message if you try to submit an approval parameter containing an ame_util.fieldDelimiter (a semicolon).

The serialization routines can accept and return strings up to the length of the data type ame_util. longestStringType, so make sure your handlers declare local variables of this type for serialized arguments.

ame_util (De)serialization Routines

```
serializeApproversTable
serializeIdList
serializeParametersTable
serializeStringList
deserializeApproversTable
deserializeIdList
deserializeParametersTable
deserializeStringList
```

Setting Values in approversTable Output Arguments

Various input and output arguments are serialized variables of type ame_util.approverRecord or ame_util.approversTable. The former is declared as follows:

```
type approverRecord is record(
  user_id integer,
  person_id integer,
  first_name varchar2(20),
  last_name varchar2(40),
  api_insertion varchar2(1),
  authority varchar2(1),
  approval_status varchar2(50));
```

The latter is a PL/SQL table of ame_util.approverRecord records. See The AME API: page - 159 for details about the record's allowed values and semantics.

When a handler creates an approverRecord, it should fetch the approver's user_id, person_id, first_name, and last_name values from fnd_user and per_all_people_f, and insert them into the record. Your code can call ame_engine.getApproverNames to fetch the first and last names. This procedure may return null in any of its output arguments, depending on which values are non-null in the fnd_user and per_all_people_f tables. The handler should set approval_status to null. How your handler sets the api_insertion and authority members is up to you. In a list-creation handler, you should set api_insertion to ame_util.AMEGenerated and authority to ame_util.authorityApprover as a rule. In a list-modification handler, you should carry the api_insertion and authority values from an approver found in *oldApproversIn* to your insertion. See The AME API: page - 159 for details regarding how the AME engine treats api_insertion and authority values.

Raising Exceptions

Your handler code should always include at least a handler for the others exception, and that exception handler should look like this:

```
exception
  when others then
   ame_util.runtimeException(packageNameIn => '[package
name]',
                              routineNameIn => '[routine
name]',
                              exceptionNumberIn => sqlcode,
                              exceptionStringIn => sqlerrm),
                             transactionIdIn =>
                               ame_engine.tempTransactionID
                             applicationIdIn =>

ame_engine.tempAmeApplicationID,
                              localErrorIn => false);
    raise;
```

If you want to raise an application-specific exception, your code should catch the exception, handle it, and re-raise it, like this:

```
errorCode integer;
errorMessage varchar2(100);
begin
/* etc. */
exception
    when [application-specific exception] then
      errorCode := -20001;
      errorMessage := '[your error message]';
      ame_util.runtimeException(packageNameIn => '[package
name]',
                                routineNameIn => '[routine
name]',
                                exceptionNumberIn =>
errorCode,
                                exceptionStringIn =>
sqlerrm),
                             transactionIdIn =>
                               ame_engine.tempTransactionID
                             applicationIdIn =>

ame_engine.tempAmeApplicationID,
                              localErrorIn => false);
raise_application_error(errorCode,
                             errorMessage);
```

If your code catches specific exceptions that prevent it from returning appropriate values to the AME engine, it should re-raise the exceptions, as does the sample when-others exception handler above.

Defining Approval Parameters

The interpretation of an approval's parameter varies with the handler category. The parameters of an authority approval type's approvals typically represent requirements for certain levels of authority, and the handler typically generates a chain of authority satisfying the most stringent such requirement. The parameters of list-modification approvals indicate specific types of list modifications, for example an alteration of the list to represent non-uniformities in an organization's signing-authority rules. The parameters of approval-group approval types' approvals typically identify a given approval group, and possibly indicate how to select members of the group for insertion into the approver list, or where in the list to insert them.

Ultimately, you must define the syntax and semantics rules for the parameters of your handler's approvals. Make sure you define these rules in a way that lets your handler code sort, aggregate, and interpret a set of parameters efficiently, because the AME engine always passes a handler all of the parameter's of the approvals of a given approval type as a set.

How do I Code an Authority Handler?

As the name suggests, a *list-creation* or *authority handler* identifies for AME the approvers within an authority hierarchy that constitute a transaction's chain of authority. Typical chains of authority are based on the approval authority (job level), position, or role of the transaction's requestor (submitter). This document uses the notion of a military chain of command as an example.

Sample handler code

The ame_absolute_job_level package is ideal sample code for authority handlers. You may wish to review this code before reading the remainder of this section.

Package-Naming Conventions

Each authority approval type must have its own PL/SQL package, compiled into the apps schema (or at least executable by the apps account). The package's name should have the syntax:

```
ame_authorityType_handler
```

For example, if you wanted to create an approval type based on military rank, you might name the approval type's handler package 'ame_military_rank_handler'.

Package Specification

Each authority handler package's specification must be the same, except for the package's name. Here is the syntax for the specification's create statement:

```
create or replace package ame_authorityType_handler as
  procedure getFirstApprover(parametersIn in varchar2,
                             firstApproverOut out varchar2);
  procedure getNextApprover(approverIn in varchar2,
                            parametersIn in varchar2,
                            nextApproverOut out varchar2);
  procedure hasFinalAuthority(approverIn in varchar2,
                              parametersIn in varchar2,
                              hasFinalAuthorityYNOut out
                             varchar2);
  procedure getSurrogate(approverIn in varchar2,
                         parametersIn in varchar2,
                         surrogateOut out varchar2);
end ame_authorityType_handler;
```

Replace *authorityType* with a short name for the type of authority implemented by the handler.

Common Arguments

All four procedures in the handler package's specification have the common argument *parametersIn*. This is a serialized ame_util.parametersTable of approval parameters in no particular order, one for each approval of each rule applying to the transaction. Hence the same approval's parameter may appear several times in the parameter list. The parameters represent—in no particular order—the authority requirements of the AME rules that apply to the transaction.

getFirstApprover Functionality

The getFirstApprover procedure must return in *firstApproverOut* a serialized ame_util.record, which should identify the first approver in the default chain of authority for the transaction with ID *ame_engine.tempTransactionId*. Continuing the military example, getFirstApprover would return the user or person ID of the transaction requestor's commanding officer.

getNextApprover Functionality

The getNextApprover procedure must return in *nextApproverOut* a serialized ame_util.record, which should identify the approver who follows in the chain of authority the approver identified by the serialized ame_util.approverRecord *approverIn*, given the values of the parameters in *parametersIn*. In the military example, getNextApprover would return the user or person ID of the commanding officer of the approver identified by *approverIn*.

hasFinalAuthority Functionality

The hasFinalAuthority procedure returns ame_util.booleanTrue in *hasFinalAuthorityYNOut* if the approver identified by the serialized ame_util.approverRecord *approverIn* has final authority for the transaction with ID *ame_engine.tempTransactionId*, given the values of the approval parameters in *parametersIn*—and returns ame_util.booleanFalse otherwise. So hasFinalAuthority must implement the following algorithm:

1. Identify the highest authority level required by the approval parameters in *parametersIn*.
2. Fetch the authority level of the approver identified by *approverIn*.
3. If the authority level in step 2 at least matches the authority level in step 1, return ame_util.booleanTrue.
4. Return ame_util.booleanFalse.

In the military example, each parameter would represent a rank, and *parametersIn* might consist of the unordered list {captain, general, major}. Suppose *approverIn* represented a lieutenant. Then step 1 in the above algorithm would determine that the transaction required a general's approval. Step 2 would determine that the current approver was a lieutenant. The comparison in step 3 would fail, and the procedure would return ame_util.booleanFalse in step 4.

getSurrogate Functionality

The getSurrogate procedure must return in *surrogateOut* a serialized ame_util.approverRecord that identifies the approver who should approve a given transaction instead of the presumably unresponsive approver identified by the serialized ame_util.approverRecord *approverIn*. When an application calls ame_api.updateApprovalStatus or ame_api.updateApprovalStatus to update the approval_status value of a given approver to ame_util.noResponseStatus, the AME engine calls the appropriate handler's getSurrogate procedure to identify a surrogate for the unresponsive approver. Often the surrogate will merely be the unresponsive approver's superior in the appropriate hierarchy, and will already be the next approver; and in this case, the engine will not modify the chain of authority. Otherwise, it will insert the surrogate into the chain of authority right after the unresponsive approver.

How do I Code a List-Modification Handler?

AME uses list-modification handlers to modify the list of approvers generated by one or more authority handlers. The three common types of list modifications are authority reductions, authority increases, and substitutions. AME provides default approval types for all three. While AME's architecture makes it possible for you to create others, it is more likely that you can meet your organization's business requirements merely by creating one or more specific approvals within the existing, default approval types.

Sample handler code

The ame_lm_handlers package is ideal sample code for list-modification handlers.

Procedure Syntax

A list-modification handler must be a PL/SQL procedure having a create statement with the following syntax:

```
create procedure
ame_modType_lm_handler(targetApproverIdsIn in varchar2,
                              approverTypesIn in varchar2,
                              parametersIn in varchar2);
```

targetApproverIdsIn is an ame_util.idList. approverTypesIn is a serialized ame_util.stringList. parametersIn is a serialized ame_util.parametersTable. The procedure can be a standalone procedure, but Oracle encourages you to code all of your list-modification handlers into a single package named 'ame_custom_lm_handlers'. If you do, you can give the procedure a name with the syntax:

```
        modType_handler
```

In this case, you should enter the package and procedure name in package.procedure format, when you use the Approvals tab to register the handler. In either case, *modType* represents the type of list modification that the handler implements. For example, a standalone list-modification handler that implements standard delegations within the military chain of command might be named 'ame_std_del_lm_handler', and the packaged version might be named 'ame_custom_lm_handlers.std_del_handler'.

Functionality

A list-modification handler modifies the list of approvers in *ame_engine.tempApproverList*. Each list-modification or substitution rule that has an approval of the type that uses the handler procedure passes one row in each of *targetApproverIdsIn*, *approverTypesIn*, and *parametersIn*. (The values for a given rule all have the same index in each argument.) The handler procedure should loop through these arguments, applying the following rules to each row in them (indexed by, say, **i**):

1. If *approverTypesIn(i)* = 'any_approver_user_id' (resp. 'any_approver_person_id') and *targetApproverIdsIn(i)* is a user_id (resp. person_id) in *oldApproversIn*, modify *ame_engine.tempApproverList* by performing the approval represented by the value of *parametersIn(i)*.
2. If *approverTypesIn(i)* = 'final_approver_user_id' (resp. 'final_approver_person_id') and *targetApproverIdsIn(i)* is the last user_id (resp. person_id) in *oldApproversIn*, modify *ame_engine.tempApproverList* by performing the approval represented by the value of *parametersIn(i)*.

Typically the value of parametersIn(i) determines what type of list modification the procedure should effect when one of the two rules above applies. For example, you might create a special list-modification handler that implemented a variety of standard military delegations (to an officer's executive officer or administrative assistant, for example), with an approval for each delegation. (The parameters for the example approvals might be 'XO' and 'AA'.) Your list-modification handler can also make its behavior vary with the values of the api_insertion and authority fields of the rows in *ame_engine.tempApproverList*.

How do I Code an Approval-Group Handler?

AME uses approval-group handlers to augment the list of approvers generated by one or more authority handlers and possibly modified by one or more list-modification handlers. The default approval-group handler has procedures for inserting an approval group's members before and after the (possibly modified) authority list. You might want to code a custom approval-group handler to pre- or post-insert a proper subset of an approval group, or to insert the approval group somewhere within the authority list.

Sample handler code

The ame_ag_handlers package is ideal sample code for approval-group handlers.

Procedure Syntax

An approval-group handler must be a PL/SQL procedure having a create statement with the following syntax:

```
create procedure ame_aGType_ag_handler(parametersIn in
                                           varchar2);
```

parametersIn is a serialized ame_util.parametersTable. The procedure can be a standalone procedure, but Oracle encourages you to code all of your approval-group handlers into a single package named 'ame_custom_ag_handlers'. If you do, you should give the procedure a name with the syntax:

```
    aGType_handler
```

and you should enter the package and procedure name in package.procedure format, when you use the Approvals tab to register the handler. In either case, *aGType* represents the type of approval-group approval that the handler implements. For example, a standalone approval-group handler that implements pre-authority-list clerical reviews might be named 'ame_clerical_ag_handler', and the packaged version might be named 'ame_custom_ag_handlers.clerical_handler'.

Functionality

An approval-group handler modifies the approver list in *ame_engine.tempApproverList*. Each approval-group rule that has an approval of the type that uses a handler passes one row to the handler in *parametersIn*. The handler should loop through the parameters, performing the approval-group insertions represented by them. Typically a parameter value in *parametersIn* identifies a specific approval group, and optionally determines how the handler procedure uses that group to augment the list of approvers (for example, what sort of subset of the group's members to insert, or where in the list order to do the insertion).

How do I Maintain Handler State?

Handler state is useful when a handler needs to avoid repeating a costly computation, typically when it would otherwise repeatedly calculate a sequence of values, where each value depends on the previous value. Handler states can be specific to the calculation of a transaction's chain of authority. This case is *per-transaction handler state*. Handler states can also be independent of the particular transaction; this is *per-handler handler state*. The ame_engine package provides a programming interface for maintaining both kinds of handler state. The programming interface lets a handler maintain one per-transaction handler state per transaction (naturally), and arbitrarily many kinds of handler state per handler. State values are case-sensitive, so for example the state 'first approver' differs from the state 'First Approver'.

The typical motivation for handler states is avoiding the necessity of walking up a chain of authority to re-establish a transaction's approval state. Without a means of maintaining handler state, the handler would have to start at the beginning of the chain each time it got called, in order to determine how many places in the chain it had already ascended. If the chain were five approvers long, and no redundant API calls occurred, the handler would ascend the chain as follows:

A

A B

A B C

A B C D

A B C D E

A B C D E null

That's 21 steps. Storing state, the behavior will instead be:

A setState

getState B setState

getState C setState

getState D setState

getState E setState

getState null clearState

That's six steps.

Handler-State Routines

Six ame_engine routines provide the programming interface that handlers should use to maintain per-transaction and per-handler state. The per-transaction routines are getHandlerTransState, clearHandlerTransState, and setHandlerTransState. The per-handler routines are getHandlerState, clearHandlerState, and setHandlerState.

Here are some notes about the data types and values for these routines' arguments:

1. handlerNameIn is the name of the actual handler package, is case-insensitive, and can be at most 50 characters in length. It is a required argument in all cases but one (clearHandlerTransState).
2. parameterIn is a varchar2(100). It is absent in the per-transaction interface, and optional in the per-handler interface. If a handler wishes to maintain only one kind of transaction-independent handler state, parameterIn should always be null. Otherwise, parameterIn should have a unique (case-sensitive) value corresponding to each kind of transaction-independent handler state.
3. stateIn is a varchar2(100). It defaults to null in both interfaces. This variable's value represents the actual state.

Here are brief descriptions of each routine's functionality:

```
function getHandlerTransState(handlerNameIn in varchar2)
return varchar2;
```

Returns the per-transaction handler state for the current transaction. If no state exists, returns null.

```
procedure clearHandlerTransState(handlerNameIn in varchar2
default null);
```

If *handlerNameIn* is null, clears all per-transaction handler states; otherwise, clears the transaction's handler state only for the handler named *handlerNameIn*.

```
procedure setHandlerTransState(handlerNameIn in varchar2,
                               stateIn in varchar2 default
null);
```

Sets (creates or updates) the per-transaction handler state of the current transaction to *stateIn*, for the handler named *handlerNameIn*.

```
function getHandlerState(handlerNameIn in varchar2,
                         parameterIn in varchar2 default
null)
  return varchar2;
```

Returns the per-handler handler state for the handler named *handlerNameIn* for the current transaction type and the (possibly null) parameter *parameterIn*. If no state exists, returns null.

```
procedure clearHandlerState(handlerNameIn in varchar2,
                            parameterIn in varchar2 default
null);
```

Clears all handler states for the handler named *handlerNameIn* and the (possibly null) parameter value *parameterIn*.

```
procedure setHandlerState(handlerNameIn in varchar2,
                          parameterIn in varchar2 default
null,
                          stateIn in varchar2 default null);
```

Sets (creates or updates) the per-handler handler state for the handler named *handlerNameIn* and the (possibly null) parameter *parameterIn*, for the current transaction type.

Typical Uses of Per-Transaction Handler State

The most typical use of handler state is avoiding repeatedly ascending part or all of a chain of authority, either to find the transaction's current location in the chain, or to identify the end of the chain. The first case occurs when knowing the person ID or user ID of an approver is insufficient to determine a transaction's current location in the chain of authority. In this case, the transaction state should be whatever data is minimally sufficient to calculate the next approver in the chain of authority. The getFirstApprover procedure sets this state, the getNextApprover procedure fetches and then updates this state, and the hasFinalAuthority procedure fetches this state, and also clears it upon returning ame_util.booleanTrue. "Implementation details" provides an example of the second case, where it is computationally costly to calculate the end of a chain of authority.

Typical Uses of Per-Handler Handler State

Per-handler state is generally useful when a handler needs to bookmark its location in a process that is not specific to a transaction. For example, in the position hierarchy, it is possible for several individuals to occupy a single position. The position-hierarchy handler's getFirstApprover and getNextApprover routines could cycle through these individuals by fetching and updating a handler state identified by a parameter having the syntax:

```
most_recent_position_approver:position ID
```

This state would presumably not be associated with a particular calling transaction type, so calls to the handler-state routines for this handler state would set *applicationIdIn* to null. A plausible *parameterIn* value would be 'most_recent_position_approver:99', meaning "The state value in this row is the person ID of the approver most recently selected for the position with position ID 99." If the state value is '123', the handler state would be interpreted as, "The ame_position_hierarchy_handler most recently selected person ID 123 from position ID 99."

Per-handler state can also be used to maintain multiple transaction-specific states. In this case, the parameter value would identify both the type of state and the transaction ID.

Handler-State Aging

AME includes a procedure that runs nightly, deleting from AME's per-handler-state table all data older than the default value of the purgeFrequency configuration variable (see purgeFrequency: page 114 for details). It is not possible to avoid this purging, because to do so would make it possible for handler-state data to accrue without limit. Therefore, handlers should in general avoid the assumption that a handler state will always exist, and should account for the possibility of a null state when attempting to fetch a state.

For example, suppose that it is computationally expensive for an authority handler to calculate from its parameter list which parameter represents the most stringent approval requirement. For example, suppose the parameter list names some military ranks, say:

```
Lieutenant
Sergeant
Executive officer
Colonel
```

The hasFinalAuthority routine might need to know which of these is the highest rank, in order to determine whether the rank of the approver identified by *approverIn* matches or exceeds the authority of the highest rank. (Perhaps an executive officer can occur at different levels of the hierarchy, depending on which chain of authority in the hierarchy one is ascending, that is, depending on the transaction requestor's location in the hierarchy.)

In such a case, the expensive computation would be to ascend the chain of authority starting at the requestor until the last rank named in the parameter list is named, and concluding that this last rank is the highest. A natural use of per-handler transaction state would be to store this value as a per-transaction handler state. (If the handler already stored some other value as a per-transaction handler state, it could instead store the state as a per-handler state having a parameter value that identified the transaction.) To avoid assuming that the state already exists, the hasFinalAuthority code would use the following logic:

1. Fetch the highest-required-rank transaction state.
2. If the state is null, calculate it the expensive way, and then store it.
3. If the rank of the approver identified by *approverIn* at least matches the highest required rank (fetched in step 1 or calculated in step 2), return ame_util.booleanTrue.
4. Return ame_util.booleanFalse.

Creating an Approval Type

You must have the Application Administrator responsibility to create an approval type.

" **To create an approval type:**

1. Choose the Approvals tab.
2. Choose the Add Approval Type button.
3. Enter a name for the approval type, the handler's name, and a user-friendly description of the approval type. When the approval type is for list-creation and exception rules, the description should refer to the hierarchy that the approval type ascends, or the principle by which the handler decides how high in the hierarchy to ascend.
4. Select any required attributes from the list of available required attributes. (You can select several attributes by holding down the **Ctrl** key while you select each attribute.)
5. Select the check boxes next to the rule types that can use the approval type. (Typically, if you select either of the first two rule types, you should select the other as well.)
6. Choose the Create Approval Type button.

   The new approval type now appears on the list of approval types.

Appendix C
The AME API

The AME API

This appendix documents AME's application programming interface (API). You should read this appendix only if you need to create a custom approval type, or customize (generally a custom or third-party) application to use AME to manage the application's approvals.

Types of Approvers

In AME, an approver list has three sub-lists. They are (in order of occurrence):

- Pre-approvers
- Authority approvers
- Post-approvers

Below are explanations of each type of approver.

Authority Approvers

Authority approvers are either members of a chain of authority within an organizational hierarchy, or are *ad hoc* approvers.

Chain-of-Authority Approvers

A *chain-of-authority approver* is an approver who is part of a chain of authority. Generally such an authority approver appears in the approver list because the approval rules that apply to the relevant transaction require the approver. However, the approver may also be a forwardee or a surrogate.

Ad hoc Authority Approvers

An *ad hoc* authority approver is dynamically inserted into a chain of authority by the application that owns the relevant transaction, after the chain of authority is complete.

Pre-Approvers

A *pre-approver* is an approver that precedes all chain-of-authority approvers in an approver list.

Post-Approvers

A *post-approver* is an approver that follows all chain-of-authority approvers in an approver list.

Data Types

Most of the AME data types you need to be aware of are declared in the package-header file ameoutil.pkh for the ame_util PL/SQL package. The AME API uses a few of these data types frequently. This section explains these commonly used types.

The ame_util.approverRecord Type

The AME API uses the ame_util.approverRecord record to represent an approver in an approver list. This data type has the declaration,

```
type approverRecord is record(
  user_id integer,
  person_id integer,
  first_name varchar2(20),
  last_name varchar2(40),
  api_insertion varchar2(1),
  authority varchar2(1),
  approval_status varchar2(50));
```

Below are explanations of the allowed values and semantics for each field in this type.

The user_id Field

The user_id field may contain any valid value from the fnd_user.user_id column. It may also contain the constant ame_util.multipleUserIds. This constant means that AME found several user_id values corresponding to the person_id value in the approverRecord. Finally, the user_id field may be null.

The person_id Field

The person_id field may contain any valid value from the per_all_people_f.person_id column, or it may be null.

The api_insertion Field

The api_insertion field may contain any of three constants. Here are brief descriptions of the constants' semantics:

ame_util.apiAuthorityInsertion represents an approver that has been inserted into the chain of authority, so that there is a discontinuity (jump) in the chain at this approver, in the sense that this approver is generally not above the chain of authority's previous approver in the relevant hierarchy. This api_insertion value results from the previous approver's forwarding a request for approval to the inserted approver.

ame_util.apiInsertion represents an *ad hoc* approver, that is, an approver who is not part of a chain of authority. If such an approver occurs between members of a chain of authority, the chain nevertheless continues past the *ad hoc* approver to the next approver in the relevant hierarchy.

ame_util.oamGenerated represents an approver required by the approval rules that apply to the relevant transaction.

The authority Field

The authority field may contain any of three constants. Here are brief descriptions of the constants' semantics:

ame_util.preApprover identifies a pre-approver.

ame_util.authorityApprover identifies an authority approver.

ame_util.postApprover identifies a post-approver.

The approval_status Field

The approval_status field may contain any of seven constants. Here are brief descriptions of the constants' semantics:

ame_util.approvedStatus means the approver approved the transaction without forwarding the request for approval.

ame_util.approveAndForwardStatus means the approver approved the transaction and also forwarded the request for approval.

ame_util.clearExceptionsStatus should be passed to ame_api.updateApprovalStatus or ame_api.updateApprovalStatus2 to clear a transaction's exception log from the application that owns the transaction, for example when the transaction's workflow is restarted.

ame_util.exceptionStatus is returned by ame_api routines when AME has raised an exception in the process of calculating a transaction's approver list. In this case, the application that owns the transaction may wish to stop the transaction's workflow. See Runtime Exceptions: page 116 for details.

ame_util.forwardStatus means the approver forwarded the request for approval of the relevant transaction, without approving the transaction.

ame_util.noResponseStatus means the application that owns the transaction requested the approver's approval of the transaction, and the approver has not responded to the request in a timely fashion. (AME responds to this status by inserting the approver's surrogate into the approver list, after the approver.)

ame_util.rejectStatus means the approver rejected the transaction.

A null approval_status means the approver has not yet responded to any request for approval of the transaction (possibly because the application that owns the transaction has not yet sent the approver such a request), but that the approver still has time to respond to such a request.

In AME, a transaction is approved when every approver in the transaction's *current* approver list has one of the approval_status values:

- ame_util.noResponseStatus
- ame_util.forwardStatus
- ame_util.approveAndForwardStatus
- ame_util.approveStatus

Note that the membership of a transaction's current approver list may change between calls to ame_api routines, depending on the approval status of the previous approver list's members. For example, if an approver's status gets updated to noResponsStatus, the AME engine will add the approver's surrogate to the current list, the next time an ame_api routine is called. This is one reason it is imperative that an application always call ame_api.getAllApprovers each time it needs to work with a transaction's approver list, and ame_api.getNextApprover each time it receives a response to a request for approval and needs to send a notification to the next required approver in the transaction's approver list. (Changes in a transaction's attribute values, organizational data, and applicable AME approval rules can also result in changes to the transaction's current approver list.)

The ame_util.approversTable Type

AME's API represents approver lists as arguments of type ame_util.approversTable. This data type is just a PL/SQL table of ame_util.approverList records. The table is always indexed by consecutive ascending integers starting at one.

The ame_util.insertionRecord Type

The getAvailableInsertions procedure uses the ame_util.insertionRecord record to represent a *possible* dynamic insertion. This data type has the declaration,

```
type insertionRecord is record(
  order_type varchar2(50),
  parameter stringType,
  api_insertion varchar2(1),
  authority varchar2(1),
  description varchar2(200));
```

Below are explanations of the allowed values and semantics for each field in this type.

The order_type and parameter Fields

The order field contains a string indicating the order relation that AME uses to determine the insertion's location in an approverList, each time AME calculates the approver list. The parameter field contains a value that indicates a specific instance of the order relation. Here are brief explanations of each possible value for the order_type field, with accompanying syntax and semantics rules for the parameter field:

ame_util.absoluteOrder means the insertionRecord's parameter field should be interpreted as an absolute order number. For example, if the approver should always be third in the list, the order_type should have this value, and the parameter should be '3'.

ame_util.afterApprover means the approver should always follow the approver that it initially followed. In this case the parameter field has the syntax:

```
{person_id,user_id}:n
```

where *n* is the person or user ID of the approver that the inserted approver should follow. For example, 'person_id:123' would identify the approver to follow as the one with the person ID 123.

ame_util.beforeApprover means the approver should always precede the approver that it initially preceded. In this case the parameter field has the syntax:

```
{person_id,user_id}:n
```

where *n* is the person or user ID of the approver that the inserted approver should precede. For example, 'person_id:123' would identify the approver to precede as the one with the person ID 123.

ame_util.firstAuthority means the approver should always be the first chain-of-authority approver in each chain of authority. In this case the parameter is not used.

ame_util.firstPostApprover means the approver should always be the first post-approver. In this case the parameter is not used.

ame_util.firstPreApprover means the approver should always be the first pre-approver. In this case the parameter is not used.

ame_util.lastPostApprover means the approver should always be the last post-approver. In this case the parameter is not used.

ame_util.lastPreApprover means the approver should always be the last pre-approver. In this case the parameter is not used.

The api_insertion Field

This field has the same allowed values and semantics as the api_insertion field of the ame_util.approverRecord data type. See The api_insertion Field under The ame_util.approverRecord Type: page 160 for details.

The authority Field

This field has the same allowed values and semantics as the authority field of the ame_util.approverRecord data type. See The authority Field under The ame_util.approverRecord Type: page 160 for details.

The description Field

This field contains a user-friendly description of the possible insertion represented by the insertionRecord.

The ame_util.insertionsTable Type

The ame_util.insertionsTable type is a PL/SQL table of ame_util.insertionRecord records. The getAvailableInsertions procedure uses an argument of this type to represent the set of *possible* dynamic insertions at a given location in an approver list. The table is always indexed by consecutive ascending integers starting at one.

The ame_util.orderRecord Type

The getAvailableInsertions and getAvailableOrders procedures use the ame_util.orderTypeRecord record to represent a *possible* dynamic insertion's order relation, for a given position in a given transaction's approver list. AME uses the order relation to determine where in the approver list to insert the approver, each time AME regenerates the relevant transaction's approver list. This data type has the declaration,

```
type orderRecord is record(
  order_type varchar2(50),
  parameter stringType,
  description varchar2(200));
```

The allowed values and semantics for each field in this type are the same as those of the ame_util.insertionRecord Type. See The ame_util.insertionRecord Type: page 162 for details.

The ame_util.ordersTable Type

The ame_util.ordersTable type is a PL/SQL table of ame_util.orderRecord records. The getAvailableInsertions and getAvailableOrders procedures use an argument of this type to represent the set of *possible* order relations for a dynamic insertion at a given location in a given transaction's approver list.

The table is always indexed by consecutive ascending integers starting at one.

AME API Routines

Arguments

Many of AME's API routines share certain arguments. Here are brief descriptions of each:

approverIn always identifies the approver of interest within a given transaction's approver list.

applicationIdIn is the fnd_application.application_id value (or, in the case of a third-party or custom application, pseudo-value) of the application that owns the transaction identified by the routine's *transactionIdIn* argument.

transactionIdIn identifies the transaction of interest. Its value must not contain white-space characters (space, tab, new-line, or return characters), and must not be a negative integer (AME uses negative integers internally to identify test transactions created on the AME test tab).

transactionTypeIn is the transaction-type identifier that distinguishes among an application's transaction types. See How AME Identifies Transaction Types: page 115 for details.

The remaining arguments used by AME's API are generally self-explanatory.

Formal and Functional Specifications

This section gives formal and functional specifications for each public routine in AME's PL/SQL API package, AME_API.

```
function validateApprover(approverIn in
ame_util.approverRecord)
  return boolean;
```

Returns true if the approver is a current (that is, validated) person or user, otherwise false.

Typical use: to verify if a specific approver's person ID or user ID is valid. Note that AME always returns valid approvers, so it is not necessary to validate approvers returned by AME. Instead, an application may wish to call this function to validate inserted approvers, before passing them to AME.

```
procedure clearAllApprovals(applicationIdIn in integer,
                 transactionIdIn in varchar2,
                 transactionTypeIn in varchar2 default
null);
```

Deletes all approvers from a transaction's approver list, including inserted approvers.

Typical use: to restart a transaction's approval process from scratch, ignoring any approvals that have already occurred--if for example a transaction was rejected, modified by the requestor, and then restarted.

```
procedure clearDeletion(approverIn in
ame_util.approverRecord,
                applicationIdIn in integer,
                transactionIdIn in varchar2,
                transactionTypeIn in varchar2 default null);
```

Clears a deletion of an approver previously requested via the deleteApprover or deleteApprovers API.

Typical use: Reverse a delete approver instruction

See also clearDeletions, deleteApprover, deleteApprovers.

```
procedure clearDeletions(applicationIdIn in integer,
                 transactionIdIn in varchar2,
                 transactionTypeIn in varchar2 default null);
```

The same as the clearDeletion API but in this instance clears multiple deletions.

Typical use: Reverse multiple delete approver instructions

See also clearDeletion, deleteApprover, deleteApprovers.

```
procedure clearInsertion(approverIn in
ame_util.approverRecord,
                 applicationIdIn in integer,
                 transactionIdIn in varchar2,
                 transactionTypeIn in varchar2 default null);
```

Clear the approver(s) inserted by calls to ame_api.insertApprover, ame_api.setFirstAuthorityApprover. It will also clear the approvers inserted as a result of forwarding. This will NOT clear the approval statuses and the approver deletions.

```
procedure clearInsertions(applicationIdIn in integer,
                  transactionIdIn in varchar2,
                             transactionTypeIn in varchar2 default
null);
```

The same as the clearInsertion API but in this instance clears multiple insertions.

```
procedure deleteApprover(applicationIdIn in integer,
                    transactionIdIn in varchar2,
                    approverIn in ame_util.approverRecord,
                    transactionTypeIn in varchar2 default
null);
```

Deletes a single approver from a transaction's approver list.

Note: This procedure should be called only *after* getNextApprover or getAllApprovers has been called, for a given transaction.

Typical use: To let an end user delete an approver that they previously inserted dynamically (*via* the user interface of the application that owns the transaction). Can also be used to delete a rule-generated approver, if the mandatory boolean attribute:

```
ALLOW_DELETING_RULE_GENERATED_APPROVERS
```

has the value 'true'. (Use this functionality with caution: it essentially defeats the business rules stored in AME.)

See also deleteApprovers.

```
procedure deleteApprovers(applicationIdIn in integer,
                          transactionIdIn in varchar2,
                          approversIn in
ame_util.approversTable,
                          transactionTypeIn in varchar2
default null);
```

Deletes a list of approvers from a transaction's approver list.

This procedure is a wrapper for deleteApprover. See deleteApprover for details.

```
procedure getAdminApprover(

                      applicationIdIn in integer default null,
                      transactionTypeIn in varchar2 default
null,
                      adminApproverOut out
ame_util.approverRecord);
```

Returns an ame_util.approverRecord identifying the administrative administrator for a given transaction's transaction type. This is the approver returned by getNextApprover and getAllApprovers (in the first row of its ame_util.approverTable return value) when an exception is raised while AME is calculating a transaction's approver list.

Typical use: one way an application can check whether AME has encountered an exception while processing a call to getNextApprover or getAllApprovers is to compare the routine's return value to getAdminApprover's return value. It is easier simply to check whether the return value of getNextApprover or getAllApprovers has the approval_status value ame_util.exceptionStatus.

See also getNextApprover and getAllApprovers.

```
    procedure getAllApprovers(applicationIdIn in integer,
                              transactionIdIn in varchar2,
                              transactionTypeIn in varchar2 default
null,
                              approversOut out
ame_util.approversTable);
```

Returns a transaction's current approver list, including both rule-generated and inserted approvers, in the order that their approval are required. The rows in *approversOut* are indexed by consecutive ascending integers starting at one. The approval_status values in *approversOut* reflect each approver's response to any requests for approvals to date that the application owning the transaction has passed to AME by calling updateApprovalStatus or updateApprovalStatus2.

Typical uses:

1. To display the entire approval list to an end user (either for information or to allow insertions and deletions).
2. To analyze a transaction's overall approval status.

Note that getAllApprovers should *not* be called just once and then used for approval routing in lieu of repeated calls to getNextApprover, because various data may change the appropriate approver list for the transaction, between approvals. See What Happens at Run Time: page 67 for details.

See also getNextApprover.

procedure **getAndRecordAllApprovers**(applicationIdIn in integer,
transactionIdIn in varchar2,
transactionTypeIn in varchar2 default null,
approversOut out
ame_util.approversTable);

This API is similar to ame_api.getAllApprovers. The only difference is that this call will also store the approver list in the AME temp tables.

procedure **getApplicableRules1**(applicationIdIn in integer,
transactionIdIn in varchar2,

```
                    transactionTypeIn in varchar2 default null,
                    ruleIdsOut out ame_util.idList);
```

Returns the list of applicable rules identifiers for the transaction. The rule id can be used to fetch further information about the rule including description, conditions, etc. If possible call getApproversAndRules if both the approver list and the applicable rules are required as only a single AME engine cycle is required.

```
procedure getApplicableRules2(applicationIdIn in integer,
                    transactionIdIn in varchar2,
                    transactionTypeIn in varchar2 default null,
                    ruleDescriptionsOut out ame_util.stringList);
```

Returns a list of rule descriptions for the transaction. The rule id is not returned only the description. If further information is required such as the applicable conditions it is necessary to call either getApplicableRules1 [or 3] or getApproversAndRules1 [or 3].

```
procedure getApplicableRules3(applicationIdIn in integer,
                    transactionIdIn in varchar2,
                    transactionTypeIn in varchar2 default null,
                    ruleIdsOut out ame_util.idList,
                    ruleDescriptionsOut out ame_util.stringList);
```

Returns a list of rule descriptions and Ids for the transaction. If the approver list is also required it is more efficient to call getApproversAndRules3 as this only requires a single AME engine cycle.

```
procedure getApproversAndRules1(applicationIdIn in integer,
                     transactionIdIn in varchar2,
                     transactionTypeIn in varchar2 default null,
                     approversOut out ame_util.approversTable,
                     ruleIdsOut out ame_util.idList);
```

Returns both applicable rules and the approver list in a single call. This procedure and its two siblings only incur a single AME engine cycle, whereas calling getApplicableRules and then getAllApprovers requires two engine cycles. Using the list of rule identifiers it is possible to fetch the relevant rule details using the getRuleDetails collection of APIs

Typical use: Fetching rule information as well as the approver list. The rule information is sometimes used to give more detailed feedback to the approver as to why the transaction requires approval

See also getRuleDetails[1-3], getApproversAndRule2 and 3

```
procedure getApproversAndRules2(applicationIdIn in integer,
                    transactionIdIn in varchar2,
                    transactionTypeIn in varchar2 default null,
                    approversOut out ame_util.approversTable,
                    ruleDescriptionsOut out ame_util.stringList);
```

Similar to getApproversAndRules1 but returns a list of rule descriptions rather than the rule Ids. Use this API if the rule description is sufficient detail.

See also getApproversAndRule1 and 3

```
procedure getApproversAndRules3(applicationIdIn in integer,
                    transactionIdIn in varchar2,
                    transactionTypeIn in varchar2 default null,
                    approversOut out ame_util.approversTable,
                    ruleIdsOut out ame_util.idList,
                    ruleDescriptionsOut out ame_util.stringList);
```

Similar to getApproversAndRules1 but in addition passes a list of the applicable rule descriptions.

See also getRuleDetails[1-3], getApproversAndRule1 and 2

```
procedure getAvailableInsertions(applicationIdIn in integer,
                                 transactionIdIn in
varchar2,
                                 positionIn in integer,
                                 transactionTypeIn in
varchar2
                                   default null,
                                 availableInsertionsOut out
nocopy

ame_util.insertionsTable);
```

Returns a list of ame_util.insertionRecord records representing the dynamic approver insertions that are possible at the absolute position *positionIn* in the transaction's current approver list. (See The ame_util.insertionRecord: page 162 for details about the interpretation of the records in *availableInsertionsOut*.)

Typical use: an application should call this procedure to get the list of dynamic insertions allowed at a given position of a given transaction's current approver list, typically to display the descriptions in the list to an end user at a user's request, to let the user select an insertion. If the user selected one of the insertions, the application would pass it to AME by calling insertApprover.

See also insertApprover and setFirstAuthorityApprover.

```
procedure getAvailableOrders(applicationIdIn in integer,
                             transactionIdIn in varchar2,
                             positionIn in integer,
                             transactionTypeIn in varchar2
default null,
                             availableOrdersOut out
                               ame_util.ordersTable);
```

Returns a list of ame_util.orderType records, including in each a user-friendly description. Each orderType record indicates an order type for a possible dynamic approver insertion at the absolute position *positionIn* in the transaction's current approver list.

Typical use: To facilitate dynamic approver insertions.

Note: this procedure is deprecated. Oracle encourages you to use getAvailableInsertions instead.

See also getAvailableInsertions.

```
procedure getNextApprover(applicationIdIn in integer,
                          transactionIdIn in varchar2,
                          transactionTypeIn in varchar2
default null,
                          nextApproverOut out
ame_util.approverRecord);
```

Returns the next approver required for the transaction, or ame_util.emptyApproverRecord when the transaction's approval process is complete.

Typical use: an application should call getNextApprover and updateApprovalStatus (or updateApprovalStatus2) iteratively for a given transaction until getNextApprover returns an ame_util.emptyApproverRecord, indicating that the transaction's approval process is complete.

See also getAllApprovers.

```
procedure getOldApprovers(applicationIdIn in integer,
                          transactionIdIn in varchar2,
                          transactionTypeIn in varchar2
default null,
                          oldApproversOut out
ame_util.approversTable);
```

Returns the approver list that AME calculated the last time such a calculation was necessary to respond to a call to AME's API. This "old" approver list is not necessarily the transaction's current list, which is what getAllApprovers returns. The old list may very well differ from the current list (that is, be incorrect!), so getOldApprovers is a deprecated routine. See What Happens at Run Time: page 67 to learn what circumstances can change a transaction's approver list between calls to AME's API.

Typical use: none. Oracle strongly encourages applications to rely instead on getAllApprovers.

See also getAllApprovers.

```
procedure getRuleDetails1(ruleIdIn in integer,
                          ruleTypeOut out varchar2,
                          ruleDescriptionOut out varchar2,
                          conditionIdsOut out ame_util.idList,
                          approvalTypeNameOut out
                varchar2,
                          approvalTypeDescriptionOut out
                varchar2,
                          approvalDescriptionOut out
                varchar2);

procedure getRuleDetails2(ruleIdIn in integer,
            ruleTypeOut out varchar2,
            ruleDescriptionOut out varchar2,
            conditionDescriptionsOut out
ame_util.longestStringList,
            approvalTypeNameOut out varchar2,
            approvalTypeDescriptionOut out varchar2,
            approvalDescriptionOut out varchar2);

  procedure getRuleDetails3(ruleIdIn in integer,
            ruleTypeOut out varchar2,
            ruleDescriptionOut out varchar2,
            conditionIdsOut out ame_util.idList,
            conditionDescriptionsOut out
ame_util.longestStringList,
            conditionHasLOVsOut out ame_util.charList,
            approvalTypeNameOut out varchar2,
            approvalTypeDescriptionOut out varchar2,
            approvalDescriptionOut out varchar2);
```

```
procedure initializeApprovalProcess(applicationIdIn in
integer,
                         transactionIdIn in varchar2,
                         transactionType in varchar2 default
null,
recordApproverListIn in boolean default false);
```

Records the date at which the transaction's approval process was initiated and optionally records the current approver list (for the sake of making that data available to getOldApprovers).

The procedure getAllApprovers requires a commit if there has been no previous call to any of the API which require transaction management. Otherwise, it is now a "read only" procedure (it does not require a commit or rollback). To make sure it functions as such, make sure your application calls

initializeApprovalProcess (with or without recording the approver list, it doesn't matter) before it calls any other ame_api routine.

Typical use: to initialize the approval process.

```
procedure getConditionDetails(conditionIdIn in integer,
             attributeNameOut out varchar2,
             attributeTypeOut out varchar2,
             attributeDescriptionOut out varchar2,
             lowerLimitOut out varchar2,
             upperLimitOut out varchar2,
             includeLowerLimitOut out  varchar2,
             includeUpperLimitOut out  varchar2,
             currencyCodeOut out varchar2,
             allowedValuesOut out
ame_util.longestStringList)
```

Retrieves the details of a condition. This API is used in conjunction with the getRuleDetails1, getRuleDetails3 API. Displaying which conditions were applicable to a rule can give valuable feedback to the approver of a transaction.

Typical use: Inform the approver what conditions were evaluated for the transaction requiring approval. Required the condition id to be passed which may be obtained from other API calls as detailed below.

See also getRuleDetails1, getRuleDetails3

```
procedure insertApprover(applicationIdIn in integer,
                         transactionIdIn in varchar2,
                         approverIn in
ame_util.approverRecord,
                         positionIn in integer,
                         orderIn in ame_util.orderRecord,
                         transactionTypeIn in varchar2
default null);
```

Dynamically inserts an approver with a given insertion-order relation at a given position in the transaction's current approver list. The procedure will succeed only under the following conditions:

1. The approval_status field of *approverIn* is null.
2. The approver identified by *approverIn* is not already in the transaction's approver list.
3. The combination of values in *orderIn*, the api_insertion field of *approverIn*, and the authority field of *approverIn* match a record returned by getAvailableInsertions.

Typical use: to insert dynamically an approver selected by an end user.

See also getAvailableInsertions and getAvailableOrders.

```
procedure setFirstAuthorityApprover(applicationIdIn in
integer,
                                    transactionIdIn in
varchar2,
                                    approverIn in

ame_util.approverRecord,
                                    transactionTypeIn in
varchar2
                                      default null);
```

Sets the first approver for *each* chain of authority in the transaction's approver list. (That is, if there are several chains of authority, they will all start from *approverIn*, even though this approver only appears at the start of the first chain.) This procedure will succeed only under the following conditions:

1. The approval_status field of *approverIn* is null.
2. The approver identified by *approverIn* is not already in the transaction's approver list.
3. No chain-of-authority approver in the approver list has a non-null approval_status value.

Typical use: to specify the beginning of a transaction's chain of authority, in the case of transaction types for which customers will generally only implement a single chain of authority.

See also insertApprover.

```
            procedure updateApprovalStatus(applicationIdIn in integer,
                                  transactionIdIn in varchar2,
                                  approverIn in ame_util.approverRecord,
                                  transactionTypeIn in varchar2 default
null,
                                  forwardeeIn in ame_util.approverRecord
                                    default ame_util.emptyApproverRecord);
```

Updates an approver's status (to the approval_status value in *approverIn*); and, if the approval_status value indicates that a forwarding has occurred, identifies the forwardee. However, if the approval_status value is ame_util.clearExceptionsStatus, the procedure clears the transaction's exception log in AME, without changing any approver's status, regardless of the approver identified by *approverIn*.

When a chain-of-authority approver forwards, AME makes the forwardee also a chain-of-authority approver. Otherwise, the forwardee has the api_insertion value ame_util.apiInsertion, and the same authority value as the forwarder.

Typical use: to update an approver's record with the approval result returned by Workflow to the application that owns the transaction, in response to a request-for-approval notification.

See also updateApprovalStatus2.

```
procedure updateApprovalStatus2(applicationIdIn in integer,
                                transactionIdIn in varchar2,
                                approvalStatusIn in
varchar2,
                                approverPersonIdIn in
integer
                                  default null,
                                approverUserIdIn in integer
default null,
                                transactionTypeIn in
varchar2
                                  default null,
                                forwardeeIn in
ame_util.approverRecord
                                  default
ame_util.emptyApproverRecord);
```

This is a wrapper for updateApprovalStatus that lets you identify an approver by person ID or user ID, rather than passing an entire ame_util.approverRecord to the API.

Typical use: to update an approver's record with the approval result returned by Workflow to the application that owns the transaction, in response to a request-for-approval notification.

See also updateApprovalStatus.

How Should a Workflow use the AME API to Manage Approvals?

The basic imperative that drives AME's design is to move all approvals-related logic out of a transaction-processing application and into AME. This logic includes checking whether an approver is, in any of three plausible senses, a "final" approver.

The Basic Algorithm

Here is the basic algorithm that an application would implement in a transaction type's workflow, to use AME to manage the transaction type's approval processes:

1. A user submits a new transaction to the application.
2. If you wish to force the chain of approval to start at an approver other than the approver that the rules will identify, call setFirstAuthorityApprover.
3. Call ame_api.getNextApprover to get the person ID or user ID of the next approver required for the transaction.
4. If ame_api.getNextApprover returns an empty ame_util.approverRecord, the transaction is approved: stop.
5. Request an approval for the transaction from the approver returned by ame_api.getNextApprover in step 3.

6. When Workflow communicates to your application the approver's response to the request for approval in step 5, translate the response into one of the allowed approval_status values for the ame_util.approverRecord data type. (Use the ame_util constants, not their values.) If the response from Workflow does not translate to one of these allowed values, handle this error condition and go to step 5.
7. If the response from Workflow amounts to ame_util.rejectStatus, handle the rejection and either go to step 5 when the rejection has been handled, or stop.
8. Call ame_api.updateApprovalStatus to pass to AME the approval status from step 6. If the status is either ame_util.approveAndForwardStatus or ame_util.forwardStatus, identify the forwardee (make sure you set all of the appropriate fields in the forwardee's ame_util.approverRecord) in the forwardeeIn argument.

   If Workflow indicates that a request for approval has timed out, you may wish to move on to the next approver without waiting further. To do so, call ame_api.updateApprovalStatus, passing in *approverIn* an ame_util.approverRecord identifying the unresponsive approver, and having the approval_status value ame_util.noResponseStatus. Do not pass an ame_util.approverRecord in *forwardeeIn*; let it default to ame_util.emptyApproverRecord.

   If the unresponsive approver is not the final approver, the AME engine will generally skip the approver (because their surrogate is generally the next approver, in the case of chain-of-authority approvers), marking them as unresponsive. If the unresponsive approver's surrogate is not already in the approver list—for example, if the unresponsive approver is the final chain-of-authority approver, and their surrogate is their supervisor—AME will insert the surrogate into the approver list immediately after the unresponsive approver.
9. Go to step 3.

Important Features of the Basic Algorithm

Note the following features of the basic algorithm:

1. There are only two possible stopping points: success at step 4, and failure at step 6.
2. AME complies with the Workflow exception-handling model. See Runtime Exceptions: page 116 for details.
3. Regardless of whether a calling application uses the Workflow exception-handling model, AME returns the approverRecord returned by ame_api.getAdminApprover, with the approval_status value ame_util.exceptionStatus, whenever it must return one or more approver records and a runtime exception occurs.
4. Oracle strongly recommends that when an erroring workflow activity's status is reset, the workflow call

ame_api.updateApprovalStatus, passing it ame_util.emptyApproverRecord for the erroring transaction, but with its approval_status field set to ame_util.clearExceptionsStatus. This will *only* clear the transaction's AME-internal exception log. If the administrative approver who resets the workflow happens to be one of the erroring transaction's approvers, it will not change that approver's status. The administrative approver should receive and respond to a normal notification requesting their approval of the transaction.

5. The algorithm only uses Workflow to process notifications--requests for approval, and responses to them. It does not use Workflow to determine which approver has "final" authority (in any sense). AME determines final authority on your application's behalf. Your code knows that a transaction has been approved when (at step 4) getNextApprover returns an ame_util.emptyApproverRecord.
6. More generally, your application does not make any decisions about the approver list's membership; it leaves all such decisions to AME, so that the logic governing a transaction type's approval processes resides in a single place (the transaction type's set of rules in AME).

How Should an Application use AME's API to Insert an Approver Dynamically?

If a calling application needs to allow end users to insert approvers dynamically, it should implement the following algorithm:

1. Call getAllApprovers to get the entire the AME-generated approver list.
2. Present this list to the end user, and prompt the user to select a location in the list to insert an approver.
3. Prompt the user to query for an approver to insert at that location.
4. Call validateApprover to ensure that the approver selected by the user is a valid approver. If the approver is invalid, go to step 3.
5. Call getAvailableInsertions for the location of interest.
6. Display the list of allowed insertion types returned by getAvailableInsertions, and prompt the user to select one of them.
7. Call insertApprover, passing it the approver and available-insertion data from steps 3 and 6.

It is extremely important that the above or similar approach is adopted. Failure to call getAvailableInsertions/insertApprover could lead to the situation that approval policies set into place are ignored when the additional approvers are inserted.

Frequently Asked Questions

How can I tell who is the final approver?

AME’s architecture lets customers define post-approval groups, so that the last (final) approver is not necessarily the approver having ”final” (signing) authority for a transaction. There is (intentionally) no provision in AME to make post-approval functionality unavailable in certain calling applications lacking it in their native approvals logic, because one of the main ideas behind AME is to let a customer use the same set of rules across several calling applications, and the customer may want to use post-approval logic for all of them. Moreover, AME’s architecture lets customers define more than one chain of authority for a single transaction, so that in such cases, typically no one person would have signing authority for all of the chains of authority required by the transaction’s approval rules. Again, this is functionality that needs to be generally available.

So in AME, not only is the notion of a ”final” approver equivocal, it is not always guaranteed to have a non-empty value, unless one defines the concept to mean simply the last approver required by the rules. In this case, a calling application determines whether an approver is the final approver by trying to fetch the next approver, and discovering that there isn’t one.

Why call AME after each approver?

It is possible for a calling application to use AME to generate an initial list of approvers, and then avoid using AME to check approval logic each time an approver responds to a request for approval. This approach short-circuits the basic algorithm (see The Basic Algorithm: page 171), and is very ill-advised, for five reasons:

1. AME accounts for the possibility that a transaction’s attributes may change during the approval process, which might result in a change of the list of rules applying to the transaction, and so to a change in the transaction’s approver list.
2. AME accounts for the possibility that the rules applying to a transaction may change (even if the transaction’s attribute values don’t change), which again would change the approver list.
3. AME accounts for organizational changes during the approval process that could affect the approver list.
4. By avoiding using AME to manage approvals decisions, an application defeats two of AME’s purposes: eliminating or avoiding replication of approvals-logic functionality in each application requiring approvals logic, and eliminating or

avoiding variations in approvals functionality across applications.

5. AME can insert a surrogate approver into a transaction's approver list, if an approver does not respond to a request for approval in a timely fashion. But using the surrogate-approver functionality requires that the application use the basic algorithm, to make sure the surrogate appears in the proper location in the approver list.

How do I force an approver who has already approved a transaction to approve it again?

An application may want an approver to re-approve a transaction, if (for example) the transaction's attribute values change. To do so, call ame_api.updateApprovalStatus or updateApprovalStatus2, identifying the approver in the appropriate argument(s) and passing a null approval_status value.

Does AME Perform any Transaction Management?

AME does *not* perform any commits or rollbacks while responding to an API call originating within a workflow. (AME determines whether it has been called from within a workflow by checking the value of the useWorkflow configuration variable.) If the calling application does not use Workflow, the only transaction management that AME performs at run time is to commit certain autonomous transactions related to its own exception log. This means that if an application calls AME's API from outside of a workflow, it must perform a commit when AME responds normally to an API call, and a rollback when AME raises an exception or returns an ame_util.approverRecord with the approval_status value ame_util.exceptionStatus.

How do I force a chain of authority generated by one rule to come before another chain generated by a second rule?

See How AME Sorts Rules at Run Time: page 134 for a discussion of the problems of rule and approval-type order.