SNMP4J-Agent & AgenPro Instrumentation Guide

Code Generation is often used in the context of Model Driven Engineering (MDE). The prerequisite for effective code generation is a modeling language that expresses domain concepts effectively. For the SNMP domain such a modeling language is the Structure of Management Information (SMI). The SMI language is often mixed up with the Abstract Syntax Notation #1 (ASN.1) though being a domain specific modeling language which ASN.1 is not.

The domain model language for SNMP is SMI.

From the code generation perspective, SMIv1 and also SMIv2 could have provided more machine readable information about object dependencies and semantics. Nevertheless, the static structure of managed objects can be fully specified using SMI.

SNMP4J-Agent is an application programming interface (API) for the SNMP command-responder domain. AgenPro generates code for that API when the SNMP4J-Agent code generation template is used. The code generation template carries the knowledge about the SNMP4J-Agent API domain. Thus, AgenPro can be used with other templates to generate code for other agent APIs, but it can only generate code for the SNMP domain because it can process SMI specifications only.

AgenPro is a code generator for the SNMP domain. It can be extended and customized through code generation templates.

The process from writing the MIB module specification to code generation and implementing the instrumentation code towards a runnable agent is illustrated by Figure 1.

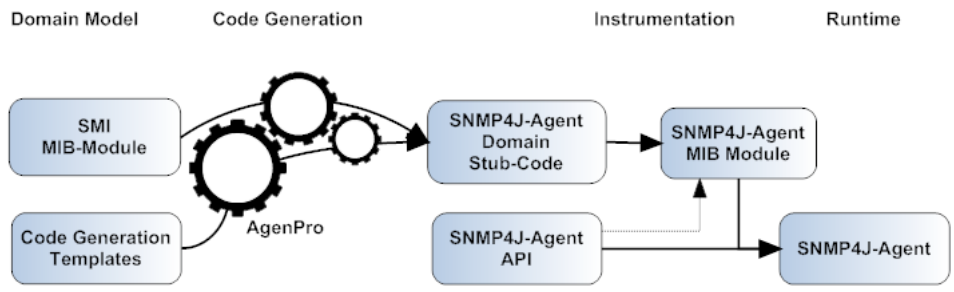

Figure 1: Schematic representation of the agent implementation process based on AgenPro code generation.

1 Instrumentation Basics

The instrumentation is the glue code connecting the managed objects generated by AgenPro with the management information data provided by a SNMP enabled (sub-)system.

For most use cases, instrumentation code cannot be generated automatically, because MIB specifications do not provide enough machine readable information about how to access management information from a specific sub-system. This is a desired characteristic of MIB specifications, because it should describe management information structure independently from a concrete system implementation.

Use customized code generation templates to generate instrumentation code for generic management information interfaces.

If a sub-system has a generic interface for accessing its management information then it might be feasible to use a customized code generation template to let it generate the instrumentation code that maps between SNMP4J-Agent API and the generic interface. A common approach to generic instrumentation is discussed in “Generic Instrumentation” on page 51.

The Java Management Extensions (JMX) framework is one example of a generic interface to management information. Other examples are proprietary CORBA interfaces and sub-agent protocols like SMUX.

In most cases a generic interface to management information is not available and the instrumentation code has to be implemented manually. How this task can be accomplished best using AgenPro in conjunction with the SNMP4J-Agent API is subject to this guide.

1.1 Management Information Classes

SNMP management information can be assigned to the following classes with their static structure illustrated by Figure 2:

- **Scalar**
  A scalar object has exactly one instance within a given context. An object identifying a sub-system, for example, is such a read-only scalar object. A mutable scalar object is a scalar object supporting read and write access, like date and time of a system.

- **Columnar**
  A columnar object type is a managed object for which none or more instances may exists in a given context. The network interface name is one example of the columnar object class. There may be none, one or more network interfaces in a sub-system. SNMP requires columnar

objects to be organized in conceptual tables where the instances of a conceptual row are identified by the same *instance identifier* also called *index*.

As with scalars, there are two types of columnar objects: immutable and mutable. If at least one columnar object is mutable, then also all conceptual row instance have to be mutable too. The simplest conceptual table implementation uses immutable columnar objects and rows.

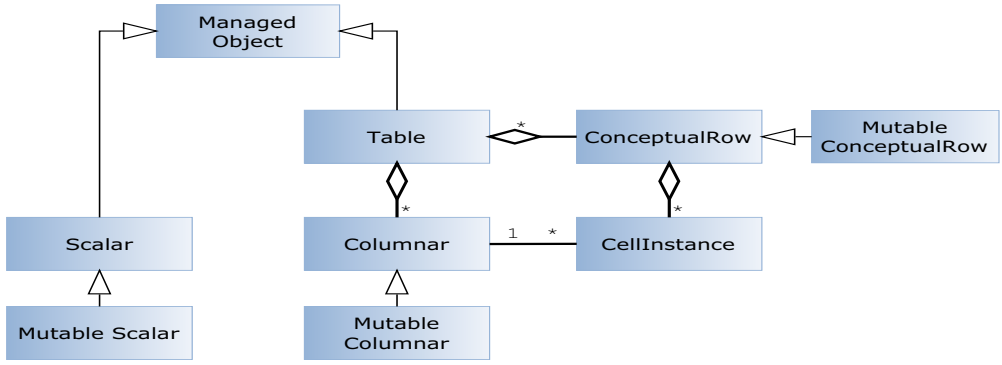

Figure 2: SNMP managed object classes as a simplified UML class diagram.

With the SNMP4J-Agent API scalar managed objects can be implemented by extending the `MOScalar` class. Alternatively and preferably, the `MOScalar` class is configured instead of extended. But configuration has its limits, especially for mutable scalars.

Conceptual SNMP Tables can be implemented by implementing the `MOTable` interface. In contrast to `MOScalar`, In contrast to the `MOScalar` class, the `MOTable` interface is not a concrete class to emphasize the conceptual background of tables in the SNMP world which gives agent developers all degrees of freedom for implementing virtual SNMP tables.

The SNMP protocol itself does not know any table concept. The table concept is part of the Structure of Management Information (SMI).

AgenPro generates instances of the `DefaultMOTable` class for SMI tables by default. For almost all use cases, the default table implementation is sufficient. It supports mutable and read-only tables as well as virtual tables, where table data is created on demand only.

For virtual table instrumentation one of the table model interfaces `MOTableModel` or `MOMutableTableModel` can be implemented. AgenPro generates instances of the `DefaultMOMutableTableModel` class by default, which is a concrete implementation of the `MOMutableTableModel` interface.

All object instances, that the code generated by the AgenPro SNMP4J-Agent templates creates, are created using a `MOFactory` instance. The default implementation of the factory (`DefaultMOFactory`) creates the default SNMP4J-Agent implementation classes. Whenever you need to create instances of your own classes (subclasses), you should not change the generated instantiation code. Instead, use your own `MOFactory` instance, which may be derived from the `DefaultMOFactory` of course.

For more information on using your own `MOFactory` see also "Generic Instrumentation" on page 51.

2 Scalar Instrumentation

2.1 Scalar Read Access

Read access on managed objects is triggered by GET, GETNEXT, and GETBULK SNMP request operations. The agent framework then calls the `ManagedObject.get(SubRequest)` method for each variable binding (sub-request) in the GET or GETNEXT SNMP PDU.

For GETBULK PDUs a sub-request does not have to correlate to variable binding in the request PDU in a one-to-one mapping. Instead, if the max-repetitions field is greater than zero, several sub-requests map to a single variable binding and vice versa. All the sub-requests that correspond to the same request variable binding are called *repetitions*.

Generally, it is sufficient to overwrite the `get` method and to set OID and value of the sub-request by calling its `SubRequest.getVariableBinding()` method.

Return always the actual management information if the information can be retrieved fast and cost effective.

For `MOScalar` objects it might be easier to overwrite `MOScalar.getValue()` to always return the actual value. The following code, for example, provides all the necessary instrumentation code for a scalar returning a system's current time as seconds since midnight, January 1, 1970 UTC:

```
public class CurrentTime extends MOScalar {
...
   public Variable getValue() {
      long sec = System.currentMillis()/1000;
      return new UnsignedInteger32(sec);
   }
}
```

If updating the value is rather expensive, then update the cache value only when management information is requested externally (e.g. via SNMP). Using this approach is also well suited if the value changes often but updating it is not very time consuming.

Overwriting the `get` method can be used to differentiate between internal read access and external read access via SNMP or other network management protocol:

```
public class CurrentTime extends MOScalar {
...
   public void get(SubRequest sreq) {
      long sec = System.currentMillis()/1000;
      setValue(new UnsignedInteger32(sec));
      super.get(sreq);
   }
}
```

The above approach updates the internal value cache each time the object's value is externally requested. The first approach does not update nor use the internal cache value provided by the `MOScalar value` member.

Besides updating a scalar's cache value when it is requested, the value can also be updated whenever the associated management information changes:

Update cache value whenever management information changes if the management information does not change very often or updating the value is time-consuming.

```
CurrentTime curTime;
...
// Update time when the system's time has changed
long sec = System.currentMillis()/1000;
setValue(new UnsignedInteger32(sec));
...
```

The decision matrix that results from the above approaches is shown by the following table:

Management Information Access Complexity	*Management Information Update Frequency*	*Instrumentation Approach*
Simple and fast	Any	Overwrite `get` method to return always actual value.
Slow (>1second)	Infrequent	Update cache value whenever managed information changes by calling `setValue`.
Slow (> 1second)	Frequently	Update cache value by overwriting `get` and calling `setValue`.

Table 1: Decision matrix for scalar read access instrumentation.

2.2 Scalar Write Access

Although implementing write access for SNMP is similar to implementing read access, there is an additional aspect that needs to be considered - it is the atomicity of SET request.

Since the variable bindings in a SET request can refer to arbitrary managed objects in an agent, a two-phase-commit (2PC) procedure is the best approach to support atomic SET request for independent managed objects. SNMP4J-Agent supports 2PC through the following methods of the `ManagedObject` interface:

The SNMP standard requires that either all variable bindings of a SET request are processed or none of them.

- `prepare` - The `prepare` method represents the commit-request (also known as preparation) phase of the 2PC procedure. For each variable binding in a SET request, SNMP4J-Agent calls the `prepare` method of each affected managed object instance. Each instance then checks whether it would be able to actually assign the

new value.
If this consistency and resource check succeeds, an undo log is written. SNMP4J-Agent supports a simple form of an undo log by providing an undo value for each sub-request in the corresponding `SubRequest` object. If the prepare phase succeeds for each variable binding, then SNMP4J-Agent continues with the `commit` phase.
Otherwise, if at least one fails, the state of the request is changed to the corresponding error status and SNMP4J-Agent continues with the `cleanup` phase.

RFC 3416 § 4.2.5 demands from implementors of the `commit` method to take all possible measures to avoid undoing a commit.

- `commit` - The `commit` method represents the commit phase of the 2PC procedure which actually alters the state of the managed object when all participating managed objects agreed to commit their changes.
  If assigning a value to a managed object fails the request's error status is set and SNMP4J-Agent then calls the `undo` method for each participating managed object.
  Otherwise, the `cleanup` method is called on all participating managed objects to finalize the request.

- `undo` - The `undo` method is being called by SNMP4J-Agent to undo committed changes using the previously saved undo log/values. Any resources allocated in `prepare` and `commit` must be freed by the `undo` method implementation.

- `cleanup` - The `cleanup` method frees up any resources allocated in the prepare (SET failed in preparation) or prepare and commit (SET succeeded) phases.

Overwriting the `setValue` method should be done with care when it is also used to set the cache value. In that case `super.setValue` should be used whenever only the cache value has to be updated.

Returning to the `CurrentTime` scalar example above, it would not make much sense to implement a 2PC for a read-write scalar version of `CurrentTime` class. Since there is no sensible validation phase if any time is allowed to be set. It would be then sufficient to implement an one phase commit only. Such an one-phase-commit instrumentation is implemented by overwriting the `setValue` method as shown by the `snmp4jAgentHBRefTime` scalar example in "SNMP4J-HEARTBEAT-MIB Instrumentation" on page 37.

A scalar object that accepts the start time for a task to be executed in the future can serve as an example for a 2PC write access instrumentation:

```
public class TaskExecutor extends DateAndTimeScalar {
...
   private Timer timer = new Timer();
...
   public void prepare(SubRequest request) {
      super.prepare(request);
      RequestStatus status = request.getStatus();
      if (status.getErrorStatus() == 0) {
         OctetString stime = (OctetString)
           request.getVariableBinding().getVariable();
         GregorianCalendar gc =
            DateAndTime.makeCalendar(stime);
         if (gc.getTime().getTime() <=
             System.currentMillis() / 1000) {
            status.setErrorStatus(PDU.wrongValue);
         }
      }
   }

   public void commit(SubRequest request) {
      super.commit(request);
      OctetString stime = (OctetString)
         request.getVariableBinding().getVariable();
      GregorianCalendar gc =
         DateAndTime.makeCalendar(stime);
      TimerTask task = createTask();
      request.setUserObject(task);
      timer.schedule(task, gc.getTime());
   }

   public void undo(SubRequest request) {
      TimerTask task =
         (TimerTask)request.getUserObject();
      if (!task.cancel()) {
         request.setErrorStatus(PDU.undoFailed);
      }
      // If not calling super.undo then do not forget
      // to set status to 'completed'!
      request.completed();
   }

   public void cleanup(SubRequest request) {
      super.cleanup(request);
   }

   private TimerTask createTask() {
      ...
}
...
}
```

This fragmentary implementation of the `TaskExecutor` class exemplifies how the preparation, commit, and undo phases can be instrumented.
By rule of thumb, call the super class' method implementation first. So you can avoid checking the value for valid type, range, etc.

In the above example, there are no resources allocated in the preparation phase. Instead the resources (i.e. the `task`) necessary to commit the SET operation are allocated in the commit phase.

In either case such resources need to be freed during the clean-up phase. The `MOScalar.cleanup` sets the `userObject` property of the `SubRequest` instance to `null`. So in many cases, there is no need to overwrite that method.

Each `SubRequest` can hold a pointer to an `Object` by its `userObject` member. In contrast to the `undoValue` member, the `userObject` is not used by the SNMP4J-Agent API and thus modifying it does not interfere the API. Modifying the `undoValue` should only be done in a direct implementation of the `ManagedObject` interface, because the SNMP4J-Agent API fills that member with the old value of the target managed object instance during the commit phase and cleans up that reference during the clean-up phase for `MOScalar` and `DefaultMOTable` instances.

3 Table Instrumentation

In contrast to a scalar value, a table may have a varying number of instances. This circumstance is taken into account by SNMP4J-Agent by dividing the responsibility for the table object instrumentation into the table and its model.

The table model implements the `MOTableModel` interface and it is responsible for providing the managed object instance values that the table manages. Basically, there are four types of table models in the SNMP context:

- **Static table models** where the number of rows do not vary or vary only within intervals of several minutes. These table models implement the `MOTableModel` interface.
  For most use cases, the concrete `DefaultMOTableModel` class can be used without modification. Although the number of rows cannot be changed directly through a SNMP SET command, static table models may support write access to their object instances. Create access, however, cannot be supported by such table models.
- **Dynamic table models** where the number of rows may vary dynamically within seconds, but as above, create access is not supported. These table models also implement the `MOTableModel` interface, but there is not a default implementation that is ready-to-use.
- **Static mutable table models** which support managed object instance (row) creation through SNMP SET in addition to static table models. Static mutable table models have to implement at least the `MOMutableTableModel` interface.
  For most use cases, the concrete `DefaultMOMutableTableModel` class can be used without modification.
- **Dynamic mutable table models** which support managed object instance (row) creation through SNMP SET while the number of rows in the model may change frequently. Like static mutable table models, the dynamic counterpart implements the `MOMutableTableModel`, however, the default implementation of that interface, the `DefaultMOMutableTableModel` class, is often not the best choice to implement a dynamic mutable table model. In many cases, a problem specific implementation of the `MOMutableTableModel` is necessary.

There are many aspects that influence the way of instrumenting SNMP tables, but the primary aspect is the combination of management information access complexity and update frequency. Whether the management information can be modified or new instances can be created through SNMP is a secondary aspect, because the added complexity (for the instrumentation) is small.

AgenPro uses instances of the `MOMutableTableModel` for all tables by default, regardless whether a table supports *read-create* access or not. When a table model implements that interface, it supports the usage of a `MOTableRowFactory` instance which is the preferred way of using custom `MOTableRow` instances as table rows. Since AgenPro generates such custom row factories for you, it uses the `MOMutableTableModel` to assign them to the used table model. If you do not want row factories or do not want to implement the mutable table model interface, then set the property `noRowFactory` to `yes` for that table object in the AgenPro configuration.

Even if using instances of the `MOMutableTableModel` interface, SNMP based row creation can be disabled by simply using only `MOColumn` instances with access *read-write* or less.

The following sections describe approaches for implementing four aspects of table instrumentation: static and dynamic tables, managed object modification and instance creation. The first two are exclusive whereas the other aspects can be combined almost arbitrarily. For an overview about the table model instrumentation approaches and how they are applied to concrete requirements see Table 2 on page 13.

3.1 Static Tables

The instrumentation approaches provided here are recommended for most use cases. Nevertheless, special requirements may reason different approaches not discussed here. So please do not treat them as law.

Static table instrumentation uses the `DefaultMOTableModel` or the `DefaultMOMutableTableModel` in conjunction with the `DefaultMOTable` class. The table model is populated with data at any of the following events:

- Initialization of the agent.
- Access to the table managed object.
- An update event from the table object instrumentation indicating that the table data has changed and thus the table model needs to be updated.

Characteristical for static table instrumentation is the passivity of the table model. Updates of the model's data are performed outside of the model

class itself. How such updates can be implemented with the default table and model implementations is shown by example in the next sections.

Management Information Access Complexity	*Management Information Update Frequency*	*Instance Modification*	*Instance Creation/ Deletion Supported*	*Instrumentation Approach*
Total table data retrival < 1s.	Any	No	No	Use `DefaultMOTableModel` with full update stragety whenever table is accessed externally. Probably use a small timeout to update the table every *n* seconds only.
Total table data retrival < 1s.	Any	Yes	No	Use `DefaultMOMutableTableModel` with full update and as rows instances of `DefaultMOMutableRow2PC` to support two-phase-commit.
Total table data retrival < 1s.	Any	Yes	Yes	Use `DefaultMOMutableTableModel` with full update as above. If SNMP index values and internal keys cannot be computed from each other a mapping table is necessary. See "Dynamic or Virtual Tables" on page 21.
Slow or expensive (> 1s per table).	Infrequent	No	No	Use `DefaultMOTableModel` with delta update strategy. See "Delta Update" on page 15.
Slow or expensive (> 1s per table).	Infrequent	Yes	No	Use `DefaultMOMutableTableModel` with delta update and as rows instances of `DefaultMOMutableRow2PC` to support two-phase-commit.
Slow or expensive (> 1s per table).	Infrequent	Yes	Yes	Use `DefaultMOMutableTableModel` with delta update as above and index-to-internal-key mapping as needed.
Slow or expensive (> 1s per table).	Frequent	Any	No	Implement `MOTableModel` interface using the virtual/dynamic table approach.

Table 2: Decision matrix for table instrumentation.

Management Information Access Complexity	*Management Information Update Frequency*	*Instance Modification*	*Instance Creation/ Deletion Supported*	*Instrumentation Approach*
Backend table with more than 10.000 rows and low memory constraint in agent.	Any	No	No	Use `BufferedMOTableModel` from SNMP4J-AgentX and implement the abstract backend data access methods in a custom subclass.
Backend table with more than 10.000 rows and low memory constraint in agent.	Any	Yes	Yes	Use `BufferedMOMutableTableModel` from SNMP4J-AgentX and implement the abstract data access methods in a custom subclass. Do not forget to add the implemented table model as row listener of the `DefaultMOTable` in order to propagate changes to the backend table implementation.

Table 2: Decision matrix for table instrumentation.

3.1.1 Initial or Full Update

When updating a table model's data, synchronization with concurrent running threads is important. Fortunately, the `DefaultMOTableModel` is completely synchronized. It is safe to add or delete rows at any time. Cell updates are also safe as long as the `DefaultMOTableRow` or derivatives thereof are used.

The following (pseudo) code illustrates an initial or full update of a table model:

```
DefaultMOMutableTableModel model =
   (DefaultMOMutableTableModel)table.getModel();
// Remove all rows from table model:
model.clean();
// Create and add the rows:
for (..) {
   Variable[] idx = ..;
   Variable[] columns = ..;
   OID index = table.getIndexDef().getIndexOID(idx);
   MOTableRow row = model.createRow(index, columns);
   model.addRow(row);
}
```

Preferably, an initial or full update should be done when the SNMP agent is offline. The update can be performed before the table is registered at the `MOServer` of the `CommandResponder` for example.

Updating the table with the above approach while the agent is listening for SNMP commands, could produce unwanted results at the command generator side. If a command is being processed by the `DefaultMOTable` instance just between the `clean` and the `addRow` calls of the update routine, then the table will be reported as being empty - which might not correctly reflect the state of the managed objects in the table.

To avoid such effects, synchronization on the table model can be used as show below:

```
synchronized (model) {
   model.clean();
   for (..) {
      // create and add rows
      ..
   }
}
```

In most cases only updating those rows that actually changed is more effective. The next section describes such an approach which is herein called *delta update*.

3.1.2 Delta Update

The delta update is characterized by updating only those rows that need to be refreshed. How do we know which rows to update? If the table has a fixed number of rows, then we could step through the table (model) and ask the instrumentation code for each row for updated data. This is no big deal and unfortunately seldom sufficient.

More often, it is necessary to detect also which rows need to be added and which to be removed. How your instrumentation code puts together that delta information is left to you. Important to know is however, when to retrieve and apply this delta information to a table's model:

1. The table's data is accessed on behalf of an external request.
2. The instrumentation code is informed (by an event or callback) from the managed system resource about an update.

Examples for external request sources are command generating entities using SNMP, AgentX, JMX, or other protocols/technologies to request management information.

The first approach is illustrated in the section "Update On External Table Access" on page 17. It is the most effective approach when management data access is relatively rare and/or the management data change rate is relatively high.

The term "frequent changes" herein denotes changes that occur once or more within the tenth of the timeout value of the data requesting entity which is normally a value between one to 60 seconds.

The second approach is less common than the first one, but can be the preferred one if the management data changes infrequently. Such a callback routine that updates a table model could look like the following pseudo code:

```
public void managedDataChanged(SomeEvent event) {
   Object key = event.getDataId();
   // map data identifier to SNMP row index
   OID index = mapKey2RowIndex(key);
   // fetch existing row - if it exists
   MOTableRow row2Update = tableModel.getRow(index);
   if (row2Update == null) {
      // create it by either using the appropriate
      // MOTableRowFactory or using the DefaultMOTable
      // - as done here:
      row2Update = table.createRow(index);
   }
   else { // update or delete it
      if (event.getData() == null) { // delete it
         tableModel.removeRow(index);
         return;
      }
   }
   // actually update the row contents
   Variable colX = event.getData().createValueX();
   row2Update.setValue(idxColX, colX);
   ..
}
```

In the above code we need no explicit synchronization, because the `addRow` and `removeRow` calls are synchronized on the `DefaultMOTableModel`. Setting the value of a row should be implemented either as an atomic operation or synchronized too. The `DefaultMOTableRow` implementation uses an atomic pointer assignment, for example.

3.1.3 Update On External Table Access

Usually the most effective way to update the data of a static table is updating just before an external request is accessing the table's data. So how can we know that a table's data is requested externally?

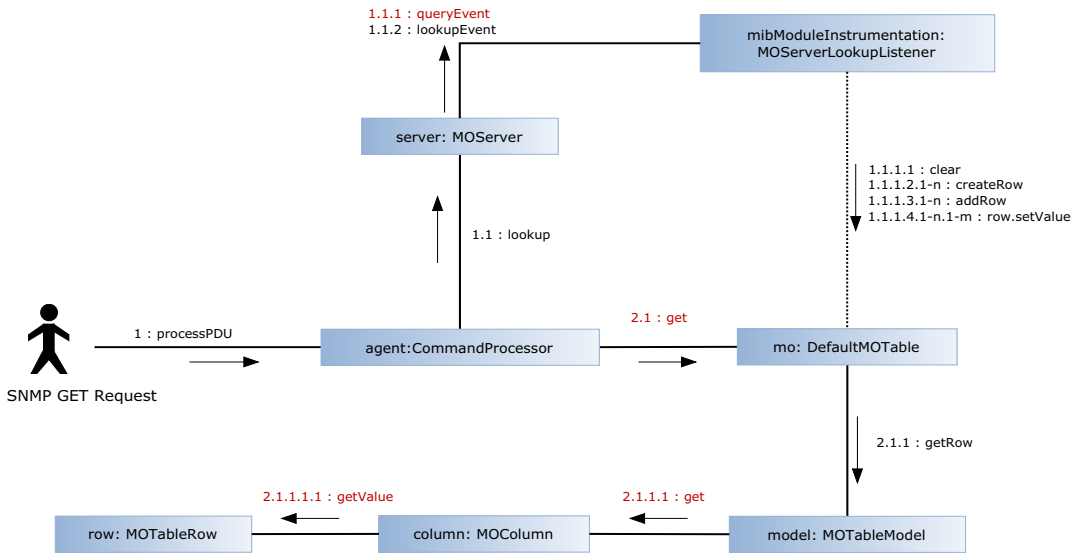

Figure 3: UML communication diagram for GET request processing with static table instrumentation and external access triggered table data updating.

Figure 3 shows by example the communication flow of a GET SNMP request and optional hookups to update the table data on-the-fly (red colored method calls). The hookups that can be used to trigger an update of tabular data can be categorized into two groups:

1. Hookups that can be used by subclassing and overwriting superclass methods.
2. Hookups that can be used by registering a listener.

Updating a table's data before it processes the request

Into the first category falls overwriting the `DefaultMOTable`'s `update` method. This method gets called by the SNMP4J-Agent API after the table object has been looked up from the `MOServer` and before a `get`, `next`, or `prepare` method is being called on behalf of an SNMP GET, GETNEXT, GETBULK or SET request respectively. It is strongly recommended to update the table model before calling any superclass method:

Although table data updates can be optimized by using the `MOScope` instance to refresh only those rows affected by the sub-request, calculating these affected regions of a table can be quite difficult, because VACM views and GETBULK repetitions as well as possible column jumps have to be taken into account.

```
public void update(MOScope updateScope) {
   // update whole table here or only those parts of
   // the table that might be affected:
   ... // do update here
   // Do not forget to call super class method finally:
   super.update(request);
}
```

Updating a table's data based on column access

If a column of a table has a prominent impact on the table's content, then an additional option is overwriting the `get` method of the `MOColumn` class or the `prepare`, `commit`, `undo`, and `cleanup` methods of the `MOMutableColumn` class respectively (see also "Managed Object Modification" on page 23). A good example for such a column is the `RowStatus` textual convention.

The `get` method of the SNMP4J-Agent `RowStatus` class shown below updates its status to `notInService` if it detects that all required columns are set (ready). This happens just before the incoming request is being answered by the super class code, which is `MOMutableColumn` in this case.

```
public void get(SubRequest subRequest,
                MOTableRow row, int column) {
   Integer32 rowStatus =
      (Integer32)getValue(row, column);
   if ((rowStatus != null) &&
       (rowStatus.getValue() == notReady)) {
      if (isReady(row, column)) {
         rowStatus.setValue(notInService);
      }

   } // Finally call super class method:
```

```
      super.get(subRequest, row, column);
   }
```

To update a table's data at instance level, the `getValue` method of the `MOTableRow` interface can be overwritten. When the `getValue` method is being called by the SNMP4J-Agent API, the `DefaultMOTable` class and the `CommandProcessor` already determined the target table cell instance. In contrast to overwriting the `get` method of the `DefaultMOTable` there is thus no need to calculate affected regions - which can be a big advantage.

In addition, implementing an `MOTableRow` or `MOMutableTableRow` instance without extending `DefaultMOTableRow`, offers the option to not cache the values of a row in the row object as the default row implementation does. The following pseudo code illustrates such a direct instrumentation:

```
public MyFanMgmtRow implements MOTableRow {
..
   public int getFanSpeed() {
      ..
   }

   public Variable getValue(int column) {
      switch(column) {
         case COL_FAN_SPEED: {
            return new Integer32(getFanSpeed());
         }
         ..
      }
   }
   ..
}
```

The approaches of the second category do not depend on subclassing. That makes them attractive in conjunction with instrumentation strategies that favor *object composition* over *class inheritance*.

SNMP4J-Agent's `CommandProcessor` uses a `MOServer` instance to lookup `ManagedObjects`. With the `MOServer`'s `addLookupListener` method an object can register a `MOServerLookupListener` that gets informed when a certain `ManagedObject` instance is about to be returned by a call to `MOServer`'s `lookup` method.

⏩ *SNMP4J-Agent tries to support object composition and class inheritance instrumentation strategies equally. You can find a description of these and other design patterns at http://en.wikipedia.org/wiki/Design_Patterns.*

Implementing the `lookupEvent` for dynamic tables is not sufficient because, for example, a dynamic table with a `DefaultMOMutableTableModel` and no rows in its cache from the models last update will not be looked up by a query because the table's `find` method will not find any instance in the table.
Therefore, the `queryEvent` has to be used to update the table model before the `find` method is executed on behalf of the query .

A `MOServerLookupListener` implements two methods: `queryEvent` and `lookupEvent`. The first is called when a `ManagedObject` is being matched against a query on behalf of the `lookup` method of the `MOServer` and the second method is called when that object actually matched the query. The `queryEvent` method should be implemented for `ManagedObjects` with dynamic content (thus changing number of object instances in its scope). Such `ManagedObjects` are dynamic tables, for example. The `lookupEvent` method should be implemented for static tables and `MOScalar` objects.

The listener is being called before any code can operate on the `ManagedObject`. The `lookup` method is suspended until all listeners return. Having these restrictions in mind, a listener can update a managed object externally. The `Snmp4jLogMib` class of SNMP4J-Agent implements such a listener:

```
public void queryEvent(MOServerLookupEvent event) {
   if ((event.getLookupResult() ==
        this.snmp4jLogLoggerEntry) &&
      (!DefaultMOQuery.isSameSource(event.getQuery(),
                        lastLoggerUpdateSource))) {
      lastLoggerUpdateSource = event.getSource();
      updateLoggerTable();
   }
}

public void lookupEvent(MOServerLookupEvent event) {
   if (event.getLookupResult()==snmp4jLogSysDescr) {
      snmp4jLogSysDescr.setValue(getLogSysDescr());
   }
   if (event.getLookupResult() == snmp4jLogFactory) {
      snmp4jLogFactory.setValue(getLogFactory());
   }
}
```

Here two scalar objects and a table object are initialized or updated when they are looked up by the `MOServer` where they have been registered. It is safe to use the object identity check to identify the looked up object, because the object reference returned by the event must be the same instance as the object that has been registered.
To avoid unnecessary table updates, the `DefaultMOQuery`'s `isSameSource` method can be used to determine whether a query is executed on behalf of the same (SNMP) request or not.

3.2 Dynamic or Virtual Tables

When you need to create a SNMP table that contains several thousand rows it could be inefficient to use static tables. With a static table all the SNMP values reside in memory although most of them are never requested.

A dynamic or virtual table only creates a SNMP view on the management information represented by the table for those rows or instances that are actually being requested. Figure 4 illustrates how a virtual table - even if row based - reduces the number of SNMP values needed in memory

while a SNMP request is being processed. In a virtual table, if there is no request processing, there will be also no SNMP values for that table in the agent's memory.

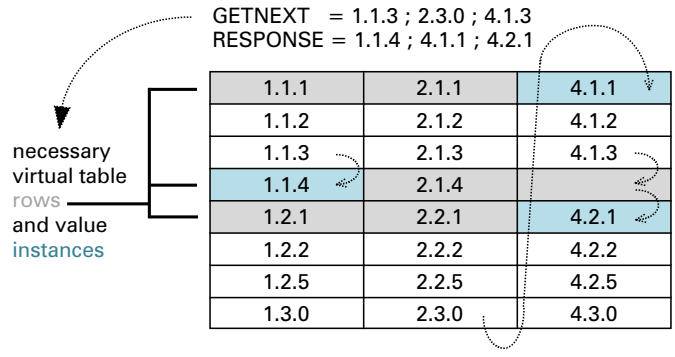

Figure 4: Virtual Table view for a GETNEXT request. The object IDs in the example are given without the table's prefix.

The computation of the cells affected by a GETNEXT or GETBULK request can be a very complex task. Fortunately, the `DefaultMOTable` class and the `CommandResponder` take over this computation task.

What is left for the virtual table instrumentation is implementing the `MOTableModel` or the `MOMutableTableModel` interface. The latter interface extends the first by means to modify the contents of the table. Crucial for the efficiency of the virtual table is, however, the implementation of the `MOTableModel`, particularly its `tailIterator` method.

The SNMP protocol supports only forward searching with its GETNEXT and GETBULK operations. Consequently, it is sufficient that the `MOTableModel` also supports forward searching only. The tail iterator starts at a specified row position in the table and then iterates until no row with lexicographic greater index value is left in the virtual table.

It is important to understand that the iterator needs to return the rows in *lexicographic index order*. If this order is not the same order by which the instrumentation code accesses the managed objects then you will have to implement an index cache that maps index values to internal keys and vice versa. If both, rows and internal keys, are equally sorted then such a cache is not necessary. Nevertheless, the instrumentation code must always be able to calculate a SNMP row index from the corresponding internal key and vice versa!

3.2.1 Buffered Table Models

The classes `BufferedMOTableModel` and `BufferedMOMutableTableModel` of SNMP4J-AgentX are abstract implementations of virtual

table models. Both classes provide a configurable internal row buffer that caches rows for an also configurable time measured in nano-seconds. The buffering of rows reduces the number of data retrieval operations to the real (backend) table.

For the `BufferedMOTableModel` the following methods need to be implemented by a subclass in order to instrument (link) it with a backend table:

METHOD SIGNATURE	DESCRIPTION
`OID firstIndex()`	Retrieves the first row index in the table as `OID`. If the table is empty `null` must be returned.
`OID lastIndex()`	Retrieves the last row index in the table as `OID`. If the table is empty `null` must be returned.
`Variable[] fetchRow(OID index)`	Fetches the row specified by its `index` from the backend table. The size of the returned array must exactly match the number of columns in the table. Otherwise, the uses `MOTableRowFactory` must be able to handle the deviation. If the row does not exist, then `null` must be returned.
`List<R> fetchNextRows(` `OID lowerBound, int chunkSize)`	Fetches a list of consecutive rows from the backend table. The row index `lowerBound` specifies the index of the first row to return. The `chunkSize` specifies the maximum number of rows to return. The rows returned must lexicographically follow each other. Each row must be created using the `rowFactory` assigned to the table model.

Table 3: The abstract methods of `BufferedMOTableModel`.

The buffered table models also buffer non existing rows that have been identified by getting a `null` return value on a `fetchRow` call.

The `BufferMOMutableTableModel` extends the `BufferedMOTableModel` and defines some additional abstract methods which are called to update the backend table.

Note: The methods that update the backend table are called only if the `rowChanged` *method of the* `MOTableRowListener` *interface has been called after the modification.*

3.3 Managed Object Modification

The two-phase-commit (2PC) protocol is a distributed algorithm which lets all participating instances agree to commit a transaction. The protocol results in either all instances committing the transaction or none.

Table write access is similar implemented as scalar write access at the instance level. Whereas scalar objects are often independently from each other mapped in a one-to-one relationship to the underlying managed objects, the instances of a table row are often conjointly mapped to a managed object. SNMP4J-Agent supports this common relationship by providing means to act on row updates. Of course, there are also means to act on table as well as on cell updates.

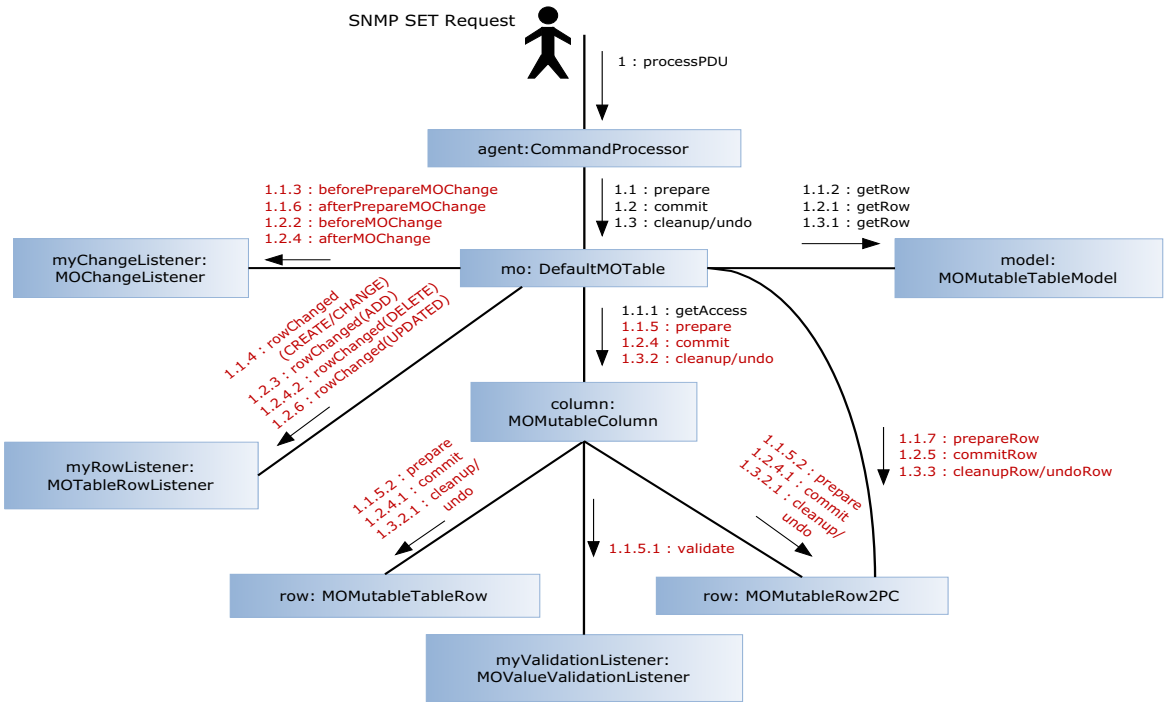

Figure 5: UML communication diagram for SET request processing on a table. The diagram changes slightly when the preparation phase fails. Then all 1.2.x messages will not be present.

Figure 5 shows the communication (i.e., call) paths of processing a SNMP SET request on a `DefaultMOTable` with at least one writable column. Analog to the read access table instrumentation, the write access instrumentation options can be categorized into two groups:

1. Options that can be used by subclassing and overwriting superclass methods:

   ⏩ Overwriting the `prepare`, `commit`, `cleanup`, and `undo` methods of the `MOMutableColumn` class.

   ⏩ Overwriting the `prepare`, `commit`, `cleanup`, and `undo` methods of the `MOMutableTableRow` or `MOMutableRow2PC` instance.

   ⏩ Overwriting the `prepareRow`, `commitRow`, `cleanupRow`, and `undoRow` methods the `MOMutableRow2PC` instance.

2. Options that can be used by registering a listener:

   ⏩ Register a `MOValueValidationListener` at the `MOMutableColumn` to check new values for consistency and conformity with an associated managed object of a column.

   ⏩ Register a `MOChangeListener` at the `MOTable` instance to accept/deny table cell updates and trigger any actions on such updates.

   ⏩ Register a `MOTableRowListener` at the `MOTable` instance to accept/deny table row updates and trigger any actions on such updates.

Particularly interesting are the options of both categories that deal with row based events. Regardless whether you use `MOTableRowEvent` or overwrite the `xxxRow` methods of a `MOMutableRow2PC` instance, you only get called once per 2PC phase of the SNMP SET request. Thus, the downstream instrumentation code can update row related managed objects at once. Because SNMP4-Agent provides a virtual row (called change set) as MOTableRow instance with these events that describes the changed columns of the row, the updates's delta information is preserved.

To be able to overwrite the `xxxRow` methods of the rows of your table, make sure that the `noRowFactory` option of AgenPro is set to "no" for your table. AgenPro then generates a `MOTableRowFactory` instance for your table which creates custom `DefaultMOMutableRow2PC` instances.

Now you can overwrite any of the `DefaultMOMutableRow2PC` methods to instrument the table rows. The `SnmpTargetMIB` of the SNMP4J-Agent framework uses this technique to update its internal tag index whenever the tag list of a row is modified:

```
public class SnmpTargetAddrEntryRow
   extends DefaultMOMutableRow2PC {

   public SnmpTargetAddrEntryRow(OID index,
```

```
                                        Variable[] values) {
      super(index, values);
      updateUserObject(this);
    }

    void updateUserObject(MOTableRow changeSet) {
      Variable tagList =
       changeSet.getValue(idxSnmpTargetAddrTagList);
      if (tagList != null) {
        Set obsolete = (Set) getUserObject();
        Set tags =
          SnmpTagList.getTags((OctetString)tagList);
        if (obsolete != null) {
          obsolete.removeAll(tags);
        }
        setUserObject(tags);
        updateIndex(obsolete, tags);
      }
    }
    public void commitRow(SubRequest subRequest,
                          MOTableRow changeSet) {
      super.commitRow(subRequest, changeSet);
      updateUserObject(changeSet);
    }
...
```

First, the tag list column's value is extracted from the change set. If it exists, it has been changed and the internal index has to be updated. The index is updated by using two lists.
The first list contains the tags to be deleted and the second list, those tags that need to be added. The obsolete list contains all the tags set before the update. Here all the rags that remain are removed to minimize changes to the index.

In SNMP4J-Agent version 1.0, only `MOTable` fired the MOChangeEvent.

Since version 1.1 of SNMP4J-Agent, the `MOChangeEvent` is fired by both `MOScalar` and `MOTable`. The `MOValueValidationEvent` is fired by both since version 1.0.

As a `MOChangeListener` you can completely control and monitor the value change of a SNMP variable (i.e., managed object instance). The only drawback is the limited view one a single object. If other objects of the same row, table, or module are not relevant then using this listener can be easier and safer than subclassing. The implementation of the `beforePrepareMOchange` of the `RowStatus` class below illustrates how this event can be used to validate a state change:

```
public void
    beforePrepareMOChange(MOChangeEvent changeEvent) {
    if (changeEvent.getOID().startsWith(oid)) {
      int currentValue = notExistant;
      if (changeEvent.getOldValue() instanceof
          Integer32) {
        currentValue = ((Integer32) changeEvent.
                        getOldValue()).getValue();
      }
      int newValue = ((Integer32) changeEvent.
                      getNewValue()).getValue();
      boolean ok = false;
```

The instrumentation code first checks whether the event is for the RowStatus column of the table. Then it gets old and new value from the event. Based on the old (current) value it checks if the state transition is allowed. If not, the change event object is modified. The event is denied by setting a SNMP (v2c/v3) error status as deny reason.

```
      switch (currentValue) {
         case notExistant:
            ok = ((newValue == createAndGo) ||
                  (newValue == createAndWait) ||
                  (newValue == destroy));
            break;
         case notReady:
            ok = ((newValue == destroy) ||
                  (newValue == active) ||
                  (newValue == notInService));
            break;
...      case destroy:
            ok = (newValue == destroy);
            break;
      }
      if (!ok) {
         changeEvent.setDenyReason(PDU.badValue);
      }
   }
}
```

MOChangeListener can be added to and removed from a table at any time. However, it is recommended to add them before registering the table at the agent's `MOServer` to avoid gaps where the listener is not active, but SNMP request are being processed.

3.4 Managed Object Creation

For scalars and tables with maximum access *read-write* or *read-only*, the number of rows and thus of managed object instances is fixed. Only tables with maximum access *read-create* support managed object instance creation. Since SNMPv2(c), the `RowStatus` textual convention is often used to define a standardized row creation and deleting mechanism. SNMP4J-Agent supports row creation with and without a `RowStatus` column in the table.

When processing a SNMP SET sub-request on a column that has *read-create* access and the row that corresponds to the object instance in the request does not exists, then `DefaultMOTable` checks if its table model implements the `MOMutableTableModel`. Then it calls the model's `createRow` method which itself uses the `MOTableRowFactory` associated with the model to actually create the new row with the default values provided by the table. Figure 6 shows the communication flow. The differences to a set request on an existing row (Figure 5) are marked red.

During the preparation phase the new row will not be added to the table. Not until the commit phase the row is added. On row creation, the row factory gets all the values assigned for the row by variables included in the

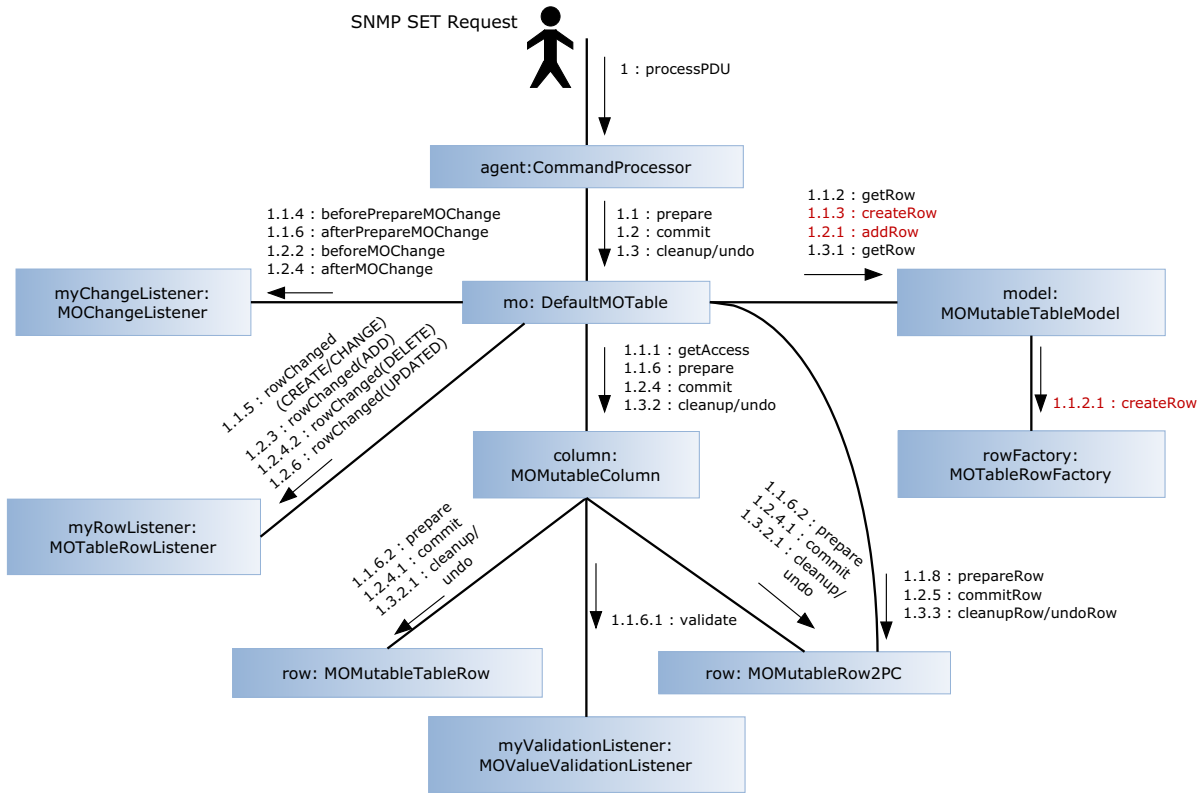

Figure 6: UML communication diagram of a successful SET request on a not existant row.

SET request and default values defined by the uninitialized columns. The factory gets always all values included in the request for that row!

Rows that can be created, normally can be deleted too. The `MOTableRowFactory` provides the `freeRow` method to allow the factory to cleanup resources it allocated during row creation. The example below from the `Snmp4jHeartbeatMib` class highlights the user added instrumentation code that initializes columns not covered by a DEFVAL specification (See "SNMP4J-HEARTBEAT-MIB" on page 31.):

```
class Snmp4jAgentHBCtrlEntryRowFactory
   extends DefaultMOMutableRow2PCFactory {
   public synchronized
      MOTableRow createRow(OID index,
                           Variable[] values)
         throws UnsupportedOperationException {
         Snmp4jAgentHBCtrlEntryRow row =
```

```
          new Snmp4jAgentHBCtrlEntryRow(index,
                                        values);
//--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::createRow
        row.setSnmp4jAgentHBCtrlLastChange(
           sysUpTime.get());
        row.setSnmp4jAgentHBCtrlEvents(
           new Counter64(0));
//--AgentGen END
        return row;
     }

   public synchronized void freeRow(MOTableRow row) {
//--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::freeRow
//--AgentGen END
   }
//--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::RowFacto-
ry
//--AgentGen END
}
```

4 Notification Instrumentation

The instrumenting notifications with the SNMP4J-Agent framework means creating the notification. The sending of the notification is done by the agent API using the `NotificationOriginator` interface. A SNMP trap or notification consists of context string and a list of variable bindings. The context string is mapped to the SNMPv3 context or to a SNMPv1/v2c community depending on the message processing model used to send the notification(s). The mapping is done by the `NotificationOriginator` implementation which is usually the `NotificationOriginatorImpl` class.

The notification originator sends notifications to zero or more targets depending on the configuration of the SNMP-NOTIFICATION-MIB and SNMP-TARGET-MIB associated with it. Thus, the instrumentation code does not send the notification directly. It does not need to handle target information. It simply forwards the payload of the notification to the notification originator which does the rest.

By default AgenPro generates a method for each NOTIFICATION-TYPE or TRAP-TYPE construct in a MIB module with the following signature:

```
public void <objectName>
   (NotificationOriginator notificationOriginator,
    OctetString context, VariableBinding[] vbs);
```

The method body serves only for checking the notification payload provided for conforming with the MIB specification. Since a notification can always carry supplemental object instances to those defined in a MIB module, the checks apply only to first *n* variable bindings, where *n* is the number of variable bindings in the notification definition. After successful check, the notification is delivered to the supplied `NotificationOriginator` for delivery to the configured targets.

The method can be called any time. Synchronization is not needed. Depending on the `NotificationOriginator` used, the method sends the notification synchronously or asynchronously. The `CommandProcessor` class, for example, uses its internal thread pool to decouple the notification instrumentation from the notification sending.

The complete example in "SNMP4J-HEARTBEAT-MIB Instrumentation" on page 37 contains on page 43 an example that shows how notification payload can be created on behalf on an external event.

As of AgenPro 3.0 it is possible to generate notification templates that directly fetch their values from the `MOServer` the module is registered with. An example notification template for the `linkDown` NOTIFICATION-TYPE which fires a `linkDown` trap for the interface with `ifIndex` 10001 is:

```
public void fireLinkDownIf10001(MOServer server,
   NotificationOriginator no,
   OctetString ctx, OID[] instanceOIDs)
{
  int numVBs = (instanceOIDs == null) ?
   3 : instanceOIDs.length;
   VariableBinding[] vbs =
     new VariableBinding[numVBs];
  {
    OID tOID = new OID("1.3.6.1.2.1.2.2.1.1.10001");
    Variable tValue =
      DefaultMOServer.getValue(server, ctx, tOID);
    vbs[1 - 1] = new VariableBinding(tOID, tValue);
  }
  {
    OID tOID = new OID("1.3.6.1.2.1.2.2.1.7.10001");
    Variable tValue =
        DefaultMOServer.getValue(server, ctx, tOID);
     vbs[2 - 1] = new VariableBinding(tOID, tValue);
  }
  {
    OID tOID = new OID("1.3.6.1.2.1.2.2.1.8.10001");
    Variable tValue =
      DefaultMOServer.getValue(server, ctx, tOID);
    vbs[3 - 1] = new VariableBinding(tOID, tValue);
  }

  if ((instanceOIDs != null) &&
     (instanceOIDs.length > 3)) {
     for (int i = 3; i < instanceOIDs.length; i++) {
       Variable tValue =
       DefaultMOServer.getValue(
        server, ctx, instanceOIDs[i]);
       vbs[i] =
       new VariableBinding(instanceOIDs[i], tValue);
     }
   }
   linkDown(no, ctx, vbs);
}
```

5 Complete Example

To illustrate MIB instrumentation with AgenPro, the following MIB definition about heartbeat event generation should serve as example. The MIB definition "SNMP4J-HEARTBEAT-MIB" on page 31 is probably a bit over engineered for a real world heartbeat event generator, but it has all ingredients necessary for showing the most important features of an AgenPro based agent instrumentation. It has a writable scalar object definition, a table with a `RowStatus` column to create and delete rows, and the MIB has a notification definition.

The generated code in the listing does not use `MOFactory` *as consequently as current versions of AgenPro SNMP4J-Agent templates do. As that code is generated, the sample is still valid and consistent and.*

The program code of the example "SNMP4J-HEARTBEAT-MIB Instrumentation" on page 37 is commented by side-heads providing some background information and instrumentation hints.

5.1 SNMP4J-HEARTBEAT-MIB

```
SNMP4J-HEARTBEAT-MIB DEFINITIONS ::= BEGIN

IMPORTS
    snmp4jAgentModules
        FROM SNMP4J-AGENT-REG
    DateAndTime,
    RowStatus,
    StorageType
        FROM SNMPv2-TC
    SnmpAdminString
        FROM SNMP-FRAMEWORK-MIB
    MODULE-IDENTITY,
    OBJECT-TYPE,
    NOTIFICATION-TYPE,
    Counter64,
    TimeTicks,
    Unsigned32
        FROM SNMPv2-SMI
    OBJECT-GROUP,
    NOTIFICATION-GROUP
        FROM SNMPv2-CONF;

snmp4jAgentHBMIB MODULE-IDENTITY
    LAST-UPDATED "200607152105Z"-- Jul 15, 2006 9:05:00 PM
    ORGANIZATION "SNMP4J.org"
    CONTACT-INFO
        "Frank Fock
         Email: fock@snmp4j.org
         Http:  www.snmp4j.org"
```

```
    DESCRIPTION
        "This example MIB module demonstrates AgenPro
        based instrumentation of a simple set of managed
        objects to manage a heart beat generator."
    REVISION "200607152105Z"-- Jul 15, 2006 9:05:00 PM
    DESCRIPTION
        "Initial version."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2
    ::= { snmp4jAgentModules 42 2 }


snmp4jAgentHBRefTime OBJECT-TYPE
    SYNTAX  DateAndTime
    MAX-ACCESS read-write
    STATUS  current
    DESCRIPTION
        "The reference time for heart-beat configurations. By
        default, the systems local time is used as reference.
        If modified, the local system's time is not changed,
        but an offset is calculated and saved to compute the
        reference time."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.1
    ::= { snmp4jAgentHBObjects 1 }


snmp4jAgentHBCtrlStartTime OBJECT-TYPE
    SYNTAX  DateAndTime
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "The time to initially start the heart-beat events.
        If not specified, the current value of
        snmp4jAgentHBRefTime is used.
        If snmp4jAgentHBCtrlDelay is greater than zero,
        the value of snmp4jAgentHBCtrlStartTime is ignored
        as if it has not been set at all."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.2
    ::= { snmp4jAgentHBCtrlEntry 2 }


snmp4jAgentHBCtrlMaxEvents OBJECT-TYPE
    SYNTAX  Unsigned32
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "The max events value specifies the maximum
        number of heartbeat events that should be
        generated on behalf of this configuration.
        The default value 0 indicates no upper limit."
    DEFVAL { 0 }
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.5
```

```
    ::= { snmp4jAgentHBCtrlEntry 5 }


snmp4jAgentHBCtrlStorageType OBJECT-TYPE
    SYNTAX  StorageType
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "The storage type for this configuration."
    DEFVAL { nonVolatile }
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.8
    ::= { snmp4jAgentHBCtrlEntry 8 }


snmp4jAgentHBCtrlDelay OBJECT-TYPE
    SYNTAX  Unsigned32
    UNITS
        "milliseconds"
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "Delay in milliseconds before the first heart-beat
        event is to be generated on behalf of this configuration.
        If this value is zero then snmp4jAgentHBCtrlStartTime
        has to be set to a date and time in the future in order to
        be able to activate the"
    DEFVAL { 1000 }
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.3
    ::= { snmp4jAgentHBCtrlEntry 3 }


snmp4jAgentHBCtrlName OBJECT-TYPE
    SYNTAX  SnmpAdminString (SIZE (1..32))
    MAX-ACCESS not-accessible
    STATUS  current
    DESCRIPTION
        "The (unique) name of the heart-beat generator."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.1
    ::= { snmp4jAgentHBCtrlEntry 1 }


snmp4jAgentHBCtrlEvents OBJECT-TYPE
    SYNTAX  Counter64
    MAX-ACCESS read-only
    STATUS  current
    DESCRIPTION
        "The number of events generated on behalf of
        this configuration since it has been last changed."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.6
    ::= { snmp4jAgentHBCtrlEntry 6 }
```

```
snmp4jAgentHBCtrlRowStatus OBJECT-TYPE
    SYNTAX  RowStatus
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "The RowStatus column."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.9
    ::= { snmp4jAgentHBCtrlEntry 9 }


snmp4jAgentExamples OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42
    ::= { snmp4jAgentModules 42 }

-- Scalars and Tables
--

snmp4jAgentHBObjects OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1
    ::= { snmp4jAgentHBMIB 1 }

snmp4jAgentHBCtrlTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF Snmp4jAgentHBCtrlEntry
    MAX-ACCESS not-accessible
    STATUS  current
    DESCRIPTION
        "The heart-beat control table contains configurations
        for heart-beat event generators."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2
    ::= { snmp4jAgentHBObjects 2 }


snmp4jAgentHBCtrlEntry OBJECT-TYPE
    SYNTAX  Snmp4jAgentHBCtrlEntry
    MAX-ACCESS not-accessible
    STATUS  current
    DESCRIPTION
        "An entry in the control table defines the parameters
        for a heart-beat event generator. A new generator
        (row) is created by setting its RowStatus column
        to createAndWait(5) or createAndGo(4)."
    INDEX {
        IMPLIED snmp4jAgentHBCtrlName }
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1
    ::= { snmp4jAgentHBCtrlTable 1 }


Snmp4jAgentHBCtrlEntry ::= SEQUENCE {

    snmp4jAgentHBCtrlName        SnmpAdminString,
```

```
    snmp4jAgentHBCtrlStartTime   DateAndTime,
    snmp4jAgentHBCtrlDelay       Unsigned32,
    snmp4jAgentHBCtrlPeriod      Unsigned32,
    snmp4jAgentHBCtrlMaxEvents   Unsigned32,
    snmp4jAgentHBCtrlEvents      Counter64,
    snmp4jAgentHBCtrlLastChange  TimeTicks,
    snmp4jAgentHBCtrlStorageType StorageType,
    snmp4jAgentHBCtrlRowStatus   RowStatus }


snmp4jAgentHBCtrlPeriod OBJECT-TYPE
    SYNTAX  Unsigned32
    UNITS
        "milli seconds"
    MAX-ACCESS read-create
    STATUS  current
    DESCRIPTION
        "The time in milli-seconds between successive
        generations of the heart-beat event."
    DEFVAL { 60000 }
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.4
    ::= { snmp4jAgentHBCtrlEntry 4 }


snmp4jAgentHBCtrlLastChange OBJECT-TYPE
    SYNTAX  TimeTicks
    MAX-ACCESS read-only
    STATUS  current
    DESCRIPTION
        "The value of sysUpTime when this configuratio
        entry has been changed. If it had been changed
        before the last system restart then zero will be
        returned."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.7
    ::= { snmp4jAgentHBCtrlEntry 7 }


-- Notification Types
--

snmp4jAgentHBEvents OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.2
    ::= { snmp4jAgentHBMIB 2 }

snmp4jAgentHBEventsID OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.2.0
    ::= { snmp4jAgentHBEvents 0 }

-- Conformance
--
```

```
snmp4jAgentHBConf OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.3
    ::= { snmp4jAgentHBMIB 3 }

-- Groups
--

snmp4jAgentHBGroups OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.3.1
    ::= { snmp4jAgentHBConf 1 }

-- Compliance
--

snmp4jAgentHBCompls OBJECT IDENTIFIER
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.3.2
    ::= { snmp4jAgentHBConf 2 }


snmp4jAgentHBEvent NOTIFICATION-TYPE
    OBJECTS {
        snmp4jAgentHBCtrlEvents}
    STATUS  current
    DESCRIPTION
        "The heart-beat event fired by a heart-beat generator."
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.2.0.1
    ::= { snmp4jAgentHBEventsID 1 }

snmp4jAgentHBBasicGroup OBJECT-GROUP
    OBJECTS {
        snmp4jAgentHBRefTime,
        snmp4jAgentHBCtrlStorageType,
        snmp4jAgentHBCtrlStartTime,
        snmp4jAgentHBCtrlRowStatus,
        snmp4jAgentHBCtrlPeriod,
        snmp4jAgentHBCtrlMaxEvents,
        snmp4jAgentHBCtrlLastChange,
        snmp4jAgentHBCtrlEvents,
        snmp4jAgentHBCtrlDelay }
    STATUS  current
    DESCRIPTION
        ""
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.3.1.1
    ::= { snmp4jAgentHBGroups 1 }

snmp4jAgentHBBasicEvents NOTIFICATION-GROUP
    NOTIFICATIONS {
        snmp4jAgentHBEvent }
    STATUS  current
    DESCRIPTION ""
    -- 1.3.6.1.4.1.4976.10.1.1.42.2.3.1.2
```

```
    ::= { snmp4jAgentHBGroups 2 }

END
```

5.2 AgenPro Project Settings

The following project settings were used to generate the stub code for the implementation shown in "SNMP4J-HEARTBEAT-MIB Instrumentation" on page 37:

```
agenpro.FileTemplate0=templates\\snmp4j-agent_1_0\\java_filename.vm
agenpro.ExecutionPerModule0=2
agenpro.InputDir0=..\\snmp4j\\src
agenpro.OutputDir0=..\\snmp4j\\src
agenpro.Modules0=SNMP4J-HEARTBEAT-MIB
agenpro.GenerationTemplate0=templates\\snmp4j-agent_1_0\\java_code.vm
agenpro.attributes.key.1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.7.0=noValueValidator
agenpro.attributes.value.1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.7.0=yes
agenpro.attributes.key.1.3.6.1.4.1.4976.10.1.1.42.2.0=constructorAccess
agenpro.attributes.value.1.3.6.1.4.1.4976.10.1.1.42.2.0=private
agenpro.attributes.key.1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.8.0=noValueValidator
agenpro.attributes.value.1.3.6.1.4.1.4976.10.1.1.42.2.1.2.1.8.0=yes
agenpro.SelectionTemplate0=templates\\snmp4j-agent_1_0\\select_1module1file.vm
```

5.3 SNMP4J-HEARTBEAT-MIB Instrumentation

```
/
*_##########################################################################
  _##
  _##  SNMP4J-Agent - Snmp4jHeartbeatMib.java
  _##
  _##  Copyright 2005-2006  Frank Fock (SNMP4J.org)
  _##
  _##  Licensed under the Apache License, Version 2.0 (the "License");
  _##  you may not use this file except in compliance with the License.
  _##  You may obtain a copy of the License at
  _##
  _##      http://www.apache.org/licenses/LICENSE-2.0
  _##
  _##  Unless required by applicable law or agreed to in writing, software
  _##  distributed under the License is distributed on an "AS IS" BASIS,
  _##  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  _##  See the License for the specific language governing permissions and
  _##  limitations under the License.
  _##

_##########################################################################*/

//--AgentGen BEGIN=_BEGIN
```

```
package org.snmp4j.agent.mo.snmp4j.example;
//--AgentGen END

import org.snmp4j.smi.*;
import org.snmp4j.mp.SnmpConstants;
import org.snmp4j.agent.*;
import org.snmp4j.agent.mo.*;
import org.snmp4j.agent.mo.snmp.*;
import org.snmp4j.agent.mo.snmp.smi.*;
import org.snmp4j.agent.request.*;
import org.snmp4j.log.LogFactory;
import org.snmp4j.log.LogAdapter;

//--AgentGen BEGIN=_IMPORT
import java.util.Timer;
import java.util.GregorianCalendar;
import java.util.Calendar;
import java.util.TimerTask;
import org.snmp4j.agent.mo.snmp4j.example.Snmp4jHeartbeatMib.
    Snmp4jAgentHBCtrlEntryRow;
import org.snmp4j.agent.mo.snmp4j.example.Snmp4jHeartbeatMib.HeartbeatTask;
import org.snmp4j.PDU;
import java.util.Date;
//--AgentGen END

public class Snmp4jHeartbeatMib
//--AgentGen BEGIN=_EXTENDS
//--AgentGen END
    implements MOGroup
//--AgentGen BEGIN=_IMPLEMENTS
    , RowStatusListener,
    MOTableRowListener
//--AgentGen END
{

  private static final LogAdapter LOGGER =
      LogFactory.getLogger(Snmp4jHeartbeatMib.class);

//--AgentGen BEGIN=_STATIC
//--AgentGen END

  // Factory
  private static MOFactory moFactory = DefaultMOFactory.getInstance();

  // Constants
  public static final OID oidSnmp4jAgentHBRefTime =
      new OID(new int[] {1, 3, 6, 1, 4, 1, 4976, 10, 1, 1, 42, 2, 1, 1, 0});
  public static final OID oidSnmp4jAgentHBEvent =
      new OID(new int[] {1, 3, 6, 1, 4, 1, 4976, 10, 1, 1, 42, 2, 2, 0, 1});
  public static final OID oidTrapVarSnmp4jAgentHBCtrlEvents =
      new OID(new int[] {1, 3, 6, 1, 4, 1, 4976, 10, 1, 1, 42, 2, 1, 2, 1, 6});
```

Add the package definition into the _BEGIN tag. AgenPro by default does not generate a package statement.

Additional import statements should be placed into the _IMPORT tag.

Additional interfaces implemented by the MIB module object can be added here. If the MIB has many objects, it can be more efficient to implement inner classes for listener objects.

The default factory can be replaced by your own factory, if you need to use your own derivations of the SNMP4J-Agent ManagedObject classes. An example use case for this would be a set of managed objects that get their management information from another system via a proprietary communication protocol.

```
  // Enumerations

  public static final class Snmp4jAgentHBCtrlStorageTypeEnum {
    /* -- eh? */
    public static final int other = 1;
    /* -- e.g., in RAM */
    public static final int _volatile = 2;
    /* -- e.g., in NVRAM */
    public static final int nonVolatile = 3;
    /* -- e.g., partially in ROM */
    public static final int permanent = 4;
    /* -- e.g., completely in ROM */
    public static final int readOnly = 5;
  }
  public static final class Snmp4jAgentHBCtrlRowStatusEnum {
    public static final int active = 1;
    /* -- the following value is a state:
-- this value may be read, but not written */
    public static final int notInService = 2;
    /* -- the following three values are
-- actions: these values may be written,
--   but are never read */
    public static final int notReady = 3;
    public static final int createAndGo = 4;
    public static final int createAndWait = 5;
    public static final int destroy = 6;
  }
```

The textual convention definitions are used by the managed object factory to determine the textual convention implementations.

```
  // TextualConventions
  private static final String TC_MODULE_SNMPV2_TC = "SNMPv2-TC";
  private static final String TC_DATEANDTIME = "DateAndTime";

  // Scalars
  private MOScalar snmp4jAgentHBRefTime;

  // Tables
  public static final OID oidSnmp4jAgentHBCtrlEntry =
      new OID(new int[] {1, 3, 6, 1, 4, 1, 4976, 10, 1, 1, 42, 2, 1, 2, 1});
```

The column sub-identifier definitions specify the column ID as defined in the corresponding MIB definition.

```
  // Column sub-identifier definitions for snmp4jAgentHBCtrlEntry:
  public static final int colSnmp4jAgentHBCtrlStartTime = 2;
  public static final int colSnmp4jAgentHBCtrlDelay = 3;
  public static final int colSnmp4jAgentHBCtrlPeriod = 4;
  public static final int colSnmp4jAgentHBCtrlMaxEvents = 5;
  public static final int colSnmp4jAgentHBCtrlEvents = 6;
  public static final int colSnmp4jAgentHBCtrlLastChange = 7;
  public static final int colSnmp4jAgentHBCtrlStorageType = 8;
  public static final int colSnmp4jAgentHBCtrlRowStatus = 9;
```

```
  // Column index definitions for snmp4jAgentHBCtrlEntry:
  public static final int idxSnmp4jAgentHBCtrlStartTime = 0;
  public static final int idxSnmp4jAgentHBCtrlDelay = 1;
  public static final int idxSnmp4jAgentHBCtrlPeriod = 2;
  public static final int idxSnmp4jAgentHBCtrlMaxEvents = 3;
  public static final int idxSnmp4jAgentHBCtrlEvents = 4;
  public static final int idxSnmp4jAgentHBCtrlLastChange = 5;
  public static final int idxSnmp4jAgentHBCtrlStorageType = 6;
  public static final int idxSnmp4jAgentHBCtrlRowStatus = 7;

  private static final MOTableSubIndex[] snmp4jAgentHBCtrlEntryIndexes =
      new MOTableSubIndex[] {
      moFactory.createSubIndex(SMIConstants.SYNTAX_OCTET_STRING, 1, 32)
  };

  private static final MOTableIndex snmp4jAgentHBCtrlEntryIndex =
      moFactory.createIndex(snmp4jAgentHBCtrlEntryIndexes,
                            true,
                            new MOTableIndexValidator() {
    public boolean isValidIndex(OID index) {
      boolean isValidIndex = true;
      //--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::isValidIndex
      //--AgentGen END
      return isValidIndex;
    }
  });

  private MOTable snmp4jAgentHBCtrlEntry;
  private MOMutableTableModel snmp4jAgentHBCtrlEntryModel;

//--AgentGen BEGIN=_MEMBERS
  private static final int[] PROTECTED_COLS =
      {
      idxSnmp4jAgentHBCtrlStartTime,
      idxSnmp4jAgentHBCtrlDelay,
      idxSnmp4jAgentHBCtrlPeriod
  };

  private Timer heartbeatTimer = new Timer();
  private int heartbeatOffset = 0;
  private NotificationOriginator notificationOriginator;
  private OctetString context;
  private SysUpTime sysUpTime;
//--AgentGen END

  private Snmp4jHeartbeatMib() {
    snmp4jAgentHBRefTime =
        new Snmp4jAgentHBRefTime(oidSnmp4jAgentHBRefTime,
                                 MOAccessImpl.ACCESS_READ_WRITE);
    snmp4jAgentHBRefTime.addMOValueValidationListener(new
        Snmp4jAgentHBRefTimeValidator());
    createSnmp4jAgentHBCtrlEntry();
```

The column index definition denotes the zero based column index within the `MOTable`.

Here additional index validation code can be added if the formal checks derived from the MIB definition are not sufficient.

This array is used to set the `mutableInService` flag to `false` for these columns to protect them against modification while the row is active.

These members are used for the instrumentation. The timer is used to trigger the heartbeat notifications which are sent by using the notification originator. Context and `sysUpTime` are needed to send the notifications. The somehow artificial `heartbeatOffset` is used to illustrate a modifiable scalar value.

Add code to the default constructor here. In most cases, these are listener registrations.

```
//--AgentGen BEGIN=_DEFAULTCONSTRUCTOR
    ((RowStatus) snmp4jAgentHBCtrlEntry.getColumn(idxSnmp4jAgentHBCtrlRowSta-
tus)).
        addRowStatusListener(this);
    snmp4jAgentHBCtrlEntry.addMOTableRowListener(this);
//--AgentGen END
  }
```

If your MIB instrumentation depends on external data, it might be necessary to implement a non-default constructor and set the access level for the default constructor to private.
This can be achieved by setting the "constructorAccess" property for the MODULE-IDENTITY object OID to "private" in the AgenPro project definition.

```
//--AgentGen BEGIN=_CONSTRUCTORS
  public Snmp4jHeartbeatMib(NotificationOriginator notificationOriginator,
                            OctetString context, SysUpTime upTime) {
    this();

    this.notificationOriginator = notificationOriginator;
    this.context = context;
    this.sysUpTime = upTime;
    // make sure that the heartbeat timer related objects are not modified
    // while row is active:
    for (int i=0; i<PROTECTED_COLS.length; i++) {
      ((MOMutableColumn)
       snmp4jAgentHBCtrlEntry.getColumn(i)).setMutableInService(false);
    }
  }

//--AgentGen END
```

The generation of the public table getter method can be deactivated by setting the "noTableGetter" property to "yes".

```
  public MOTable getSnmp4jAgentHBCtrlEntry() {
    return snmp4jAgentHBCtrlEntry;
  }

  private void createSnmp4jAgentHBCtrlEntry() {
    MOColumn[] snmp4jAgentHBCtrlEntryColumns = new MOColumn[8];
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlStartTime] =
        new DateAndTime(colSnmp4jAgentHBCtrlStartTime,
                        MOAccessImpl.ACCESS_READ_CREATE,
                        null);
    ValueConstraint snmp4jAgentHBCtrlStartTimeVC = new ConstraintsImpl();
   ((ConstraintsImpl) snmp4jAgentHBCtrlStartTimeVC).add(new Constraint(8, 8));
    ((ConstraintsImpl) snmp4jAgentHBCtrlStartTimeVC).add(new Constraint(11,
11));
    ((MOMutableColumn) snmp4jAgentHBCtrlEntryColumns[
     idxSnmp4jAgentHBCtrlStartTime]).
        addMOValueValidationListener(new ValueConstraintValidator(
        snmp4jAgentHBCtrlStartTimeVC));
    ((MOMutableColumn) snmp4jAgentHBCtrlEntryColumns[
     idxSnmp4jAgentHBCtrlStartTime]).
      addMOValueValidationListener(new Snmp4jAgentHBCtrlStartTimeValidator());
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlDelay] =
        new MOMutableColumn(colSnmp4jAgentHBCtrlDelay,
                            SMIConstants.SYNTAX_GAUGE32,
                            MOAccessImpl.ACCESS_READ_CREATE,
```

```
                          new UnsignedInteger32(1000));
    ((MOMutableColumn) snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlDe-
lay]).
        addMOValueValidationListener(new Snmp4jAgentHBCtrlDelayValidator());
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlPeriod] =
        new MOMutableColumn(colSnmp4jAgentHBCtrlPeriod,
                            SMIConstants.SYNTAX_GAUGE32,
                            MOAccessImpl.ACCESS_READ_CREATE,
                            new UnsignedInteger32(60000));
    ((MOMutableColumn) snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlPeri-
od]).
        addMOValueValidationListener(new Snmp4jAgentHBCtrlPeriodValidator());
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlMaxEvents] =
      new MOMutableColumn(colSnmp4jAgentHBCtrlMaxEvents,
                          SMIConstants.SYNTAX_GAUGE32,
                          MOAccessImpl.ACCESS_READ_CREATE,
                          new UnsignedInteger32(0));
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlEvents] =
      moFactory.createColumn(colSnmp4jAgentHBCtrlEvents,
                             SMIConstants.SYNTAX_COUNTER64,
                             MOAccessImpl.ACCESS_READ_ONLY);
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlLastChange] =
        moFactory.createColumn(colSnmp4jAgentHBCtrlLastChange,
                               SMIConstants.SYNTAX_TIMETICKS,
                               MOAccessImpl.ACCESS_READ_ONLY);
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlEvents] =
        moFactory.createColumn(colSnmp4jAgentHBCtrlEvents,
                               SMIConstants.SYNTAX_COUNTER64,
                               MOAccessImpl.ACCESS_READ_ONLY);
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlStorageType] =
        new StorageType(colSnmp4jAgentHBCtrlStorageType,
                        MOAccessImpl.ACCESS_READ_CREATE,
                        new Integer32(3));
   ValueConstraint snmp4jAgentHBCtrlStorageTypeVC = new EnumerationConstraint(
        new int[] {Snmp4jAgentHBCtrlStorageTypeEnum.other,
        Snmp4jAgentHBCtrlStorageTypeEnum._volatile,
        Snmp4jAgentHBCtrlStorageTypeEnum.nonVolatile,
        Snmp4jAgentHBCtrlStorageTypeEnum.permanent,
        Snmp4jAgentHBCtrlStorageTypeEnum.readOnly});
    ((MOMutableColumn)snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlStora-
geType]).
     addMOValueValidationListener(new ValueConstraintValidator(snmp4jAgentHBC-
trlStorageTypeVC));
    snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlRowStatus] =
        new RowStatus(colSnmp4jAgentHBCtrlRowStatus);
    ValueConstraint snmp4jAgentHBCtrlRowStatusVC = new EnumerationConstraint(
        new int[] {Snmp4jAgentHBCtrlRowStatusEnum.active,
        Snmp4jAgentHBCtrlRowStatusEnum.notInService,
        Snmp4jAgentHBCtrlRowStatusEnum.notReady,
        Snmp4jAgentHBCtrlRowStatusEnum.createAndGo,
        Snmp4jAgentHBCtrlRowStatusEnum.createAndWait,
        Snmp4jAgentHBCtrlRowStatusEnum.destroy});
```

```
    ((MOMutableColumn)snmp4jAgentHBCtrlEntryColumns[idxSnmp4jAgentHBCtrlRowSta-
tus]).
      addMOValueValidationListener(new ValueConstraintValidator(snmp4jAgentHBC-
trlRowStatusVC));

    snmp4jAgentHBCtrlEntryModel = new DefaultMOMutableTableModel();
   snmp4jAgentHBCtrlEntryModel.setRowFactory(new Snmp4jAgentHBCtrlEntryRowFac-
tory());
    snmp4jAgentHBCtrlEntry =
        moFactory.createTable(oidSnmp4jAgentHBCtrlEntry,
                              snmp4jAgentHBCtrlEntryIndex,
                              snmp4jAgentHBCtrlEntryColumns,
                              snmp4jAgentHBCtrlEntryModel);
  }
```

The `registerMOs` method must be called by the agent's main routine to register the herein defined objects at the `MOServer` of the agent.

```
  public void registerMOs(MOServer server, OctetString context) throws
      DuplicateRegistrationException {
    // Scalar Objects
    server.register(this.snmp4jAgentHBRefTime, context);
    server.register(this.snmp4jAgentHBCtrlEntry, context);
//--AgentGen BEGIN=_registerMOs
//--AgentGen END
  }

  public void unregisterMOs(MOServer server, OctetString context) {
    // Scalar Objects
    server.unregister(this.snmp4jAgentHBRefTime, context);
    server.unregister(this.snmp4jAgentHBCtrlEntry, context);
//--AgentGen BEGIN=_unregisterMOs
//--AgentGen END
  }
```

The code generated for notifications is not really required. It simply checks the notification to be sent for compliance with the MIB specification.

```
  // Notifications
  public void snmp4jAgentHBEvent(NotificationOriginator notificationOriginator,
                                 OctetString context, VariableBinding[] vbs) {
    if (vbs.length < 1) {
      throw new IllegalArgumentException("Too few notification objects: " +
                                         vbs.length + "<1");
    }
    if (!(vbs[0].getOid().startsWith(oidTrapVarSnmp4jAgentHBCtrlEvents))) {
      throw new IllegalArgumentException("Variable 0 has wrong OID: " +
                                         vbs[0].getOid() +
                                         " does not start with " +
                                         oidTrapVarSnmp4jAgentHBCtrlEvents);
    }
    if (!snmp4jAgentHBCtrlEntryIndex.isValidIndex(snmp4jAgentHBCtrlEntry.
                                                  getIndexPart(vbs[0].getOid()))) {
      throw new IllegalArgumentException(
          "Illegal index for variable 0 specified: " +
          snmp4jAgentHBCtrlEntry.getIndexPart(vbs[0].getOid()));
    }
    notificationOriginator.notify(context, oidSnmp4jAgentHBEvent, vbs);
```

```
  }

  // Scalars
  public class Snmp4jAgentHBRefTime extends DateAndTimeScalar {
    Snmp4jAgentHBRefTime(OID oid, MOAccess access) {
      super(oid, access, new OctetString());
//--AgentGen BEGIN=snmp4jAgentHBRefTime
//--AgentGen END
    }

    public int isValueOK(SubRequest request) {
      Variable newValue =
          request.getVariableBinding().getVariable();
      int valueOK = super.isValueOK(request);
      if (valueOK != SnmpConstants.SNMP_ERROR_SUCCESS) {
        return valueOK;
      }
      OctetString os = (OctetString) newValue;
      if (!(((os.length() >= 8) && (os.length() <= 8)) ||
            ((os.length() >= 11) && (os.length() <= 11)))) {
        valueOK = SnmpConstants.SNMP_ERROR_WRONG_LENGTH;
      }
      //--AgentGen BEGIN=snmp4jAgentHBRefTime::isValueOK
      //--AgentGen END
      return valueOK;
    }

    public Variable getValue() {
      //--AgentGen BEGIN=snmp4jAgentHBRefTime::getValue
      GregorianCalendar gc = new GregorianCalendar();
      gc.add(Calendar.MILLISECOND, heartbeatOffset);
      super.setValue(DateAndTime.makeDateAndTime(gc));
      //--AgentGen END
      return super.getValue();
    }

    public int setValue(Variable newValue) {
      //--AgentGen BEGIN=snmp4jAgentHBRefTime::setValue
```

The `isValueOK` method checks if the value specified in a sub-request (variable binding) is a valid one for this instance. The code generated can only check against the machine interpretable restrictions provided by the MIB specification. Additional semantic checks need to be placed into the `isValueOK` tag.

This instrumentation modifies the cache value which is the required approach when implementing AgenPro generated scalars. See also "Scalar Read Access" on page 6.

The write access instrumentation is a one-phase-commit implementation. Also the cache value is modified, the value returned on a GET request will be always recomputed by the `getValue` method. However, there could be a race condition where a SET request could modify the cache value after it has been set by another GET request, but before it has returned the cache value. To avoid such a scenario, synchronization or a true computation of the cache value in the SET method would be necessary.

```
    GregorianCalendar gc = DateAndTime.makeCalendar((OctetString) newValue);
    GregorianCalendar curGC = new GregorianCalendar();
    heartbeatOffset = (int) (gc.getTimeInMillis() - curGC.getTimeInMillis());
    //--AgentGen END
    return super.setValue(newValue);
  }

  //--AgentGen BEGIN=snmp4jAgentHBRefTime::_METHODS
  //--AgentGen END

}

// Value Validators
/**
 * The <code>Snmp4jAgentHBRefTimeValidator</code> implements the value
 * validation for <code>Snmp4jAgentHBRefTime</code>.
 */
static class Snmp4jAgentHBRefTimeValidator implements
    MOValueValidationListener {

  public void validate(MOValueValidationEvent validationEvent) {
    Variable newValue = validationEvent.getNewValue();
    OctetString os = (OctetString) newValue;
    if (!(((os.length() >= 8) && (os.length() <= 8)) ||
          ((os.length() >= 11) && (os.length() <= 11)))) {
      validationEvent.setValidationStatus(SnmpConstants.
                                          SNMP_ERROR_WRONG_LENGTH);
      return;
    }
    //--AgentGen BEGIN=snmp4jAgentHBRefTime::validate
    //--AgentGen END
  }
}

/**
 * The <code>Snmp4jAgentHBCtrlStartTimeValidator</code> implements the value
 * validation for <code>Snmp4jAgentHBCtrlStartTime</code>.
 */
static class Snmp4jAgentHBCtrlStartTimeValidator implements
    MOValueValidationListener {

  public void validate(MOValueValidationEvent validationEvent) {
    Variable newValue = validationEvent.getNewValue();
    OctetString os = (OctetString) newValue;
    if (!(((os.length() >= 8) && (os.length() <= 8)) ||
          ((os.length() >= 11) && (os.length() <= 11)))) {
      validationEvent.setValidationStatus(SnmpConstants.
                                          SNMP_ERROR_WRONG_LENGTH);
      return;
    }
    //--AgentGen BEGIN=snmp4jAgentHBCtrlStartTime::validate
```

To disable the generation of a validator for a specific MIB object, set the property "noValueValidator" for the OID of that object in AgenPro. To disable it for all objects, set the property to the `iso` node of the MIB tree.

```
    //--AgentGen END
  }
}

/**
 * The <code>Snmp4jAgentHBCtrlDelayValidator</code> implements the value
 * validation for <code>Snmp4jAgentHBCtrlDelay</code>.
 */
static class Snmp4jAgentHBCtrlDelayValidator implements
    MOValueValidationListener {

  public void validate(MOValueValidationEvent validationEvent) {
    Variable newValue = validationEvent.getNewValue();
    //--AgentGen BEGIN=snmp4jAgentHBCtrlDelay::validate
    //--AgentGen END
  }
}

/**
 * The <code>Snmp4jAgentHBCtrlPeriodValidator</code> implements the value
 * validation for <code>Snmp4jAgentHBCtrlPeriod</code>.
 */
static class Snmp4jAgentHBCtrlPeriodValidator implements
    MOValueValidationListener {

  public void validate(MOValueValidationEvent validationEvent) {
    Variable newValue = validationEvent.getNewValue();
    //--AgentGen BEGIN=snmp4jAgentHBCtrlPeriod::validate
    //--AgentGen END
  }
}

// Rows and Factories

public class Snmp4jAgentHBCtrlEntryRow extends DefaultMOMutableRow2PC {
  public Snmp4jAgentHBCtrlEntryRow(OID index, Variable[] values) {
    super(index, values);
  }

  public OctetString getSnmp4jAgentHBCtrlStartTime() {
    return (OctetString) getValue(idxSnmp4jAgentHBCtrlStartTime);
  }

  public void setSnmp4jAgentHBCtrlStartTime(OctetString newValue) {
    setValue(idxSnmp4jAgentHBCtrlStartTime, newValue);
  }

  public UnsignedInteger32 getSnmp4jAgentHBCtrlDelay() {
    return (UnsignedInteger32) getValue(idxSnmp4jAgentHBCtrlDelay);
  }

  public void setSnmp4jAgentHBCtrlDelay(UnsignedInteger32 newValue) {
```

It is recommended to use row factories for tables, since then a row class is implemented that provides convenient access methods for the columns of that row. In addition, the row class can be extended with custom functions by placing them into the `Row` tag. Row factories can be disabled by setting the "noRowFactory" property to "yes".

```
    setValue(idxSnmp4jAgentHBCtrlDelay, newValue);
  }

  public UnsignedInteger32 getSnmp4jAgentHBCtrlPeriod() {
    return (UnsignedInteger32) getValue(idxSnmp4jAgentHBCtrlPeriod);
  }

  public void setSnmp4jAgentHBCtrlPeriod(UnsignedInteger32 newValue) {
    setValue(idxSnmp4jAgentHBCtrlPeriod, newValue);
  }

  public UnsignedInteger32 getSnmp4jAgentHBCtrlMaxEvents() {
    return (UnsignedInteger32) getValue(idxSnmp4jAgentHBCtrlMaxEvents);
  }

  public void setSnmp4jAgentHBCtrlMaxEvents(UnsignedInteger32 newValue) {
    setValue(idxSnmp4jAgentHBCtrlMaxEvents, newValue);
  }

  public Counter64 getSnmp4jAgentHBCtrlEvents() {
    return (Counter64) getValue(idxSnmp4jAgentHBCtrlEvents);
  }

  public void setSnmp4jAgentHBCtrlEvents(Counter64 newValue) {
    setValue(idxSnmp4jAgentHBCtrlEvents, newValue);
  }

  public TimeTicks getSnmp4jAgentHBCtrlLastChange() {
    return (TimeTicks) getValue(idxSnmp4jAgentHBCtrlLastChange);
  }

  public void setSnmp4jAgentHBCtrlLastChange(TimeTicks newValue) {
    setValue(idxSnmp4jAgentHBCtrlLastChange, newValue);
  }

  public Integer32 getSnmp4jAgentHBCtrlStorageType() {
    return (Integer32) getValue(idxSnmp4jAgentHBCtrlStorageType);
  }

  public void setSnmp4jAgentHBCtrlStorageType(Integer32 newValue) {
    setValue(idxSnmp4jAgentHBCtrlStorageType, newValue);
  }

  public Integer32 getSnmp4jAgentHBCtrlRowStatus() {
    return (Integer32) getValue(idxSnmp4jAgentHBCtrlRowStatus);
  }

  public void setSnmp4jAgentHBCtrlRowStatus(Integer32 newValue) {
    setValue(idxSnmp4jAgentHBCtrlRowStatus, newValue);
  }
```

```
    //--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::Row
    //--AgentGen END
  }

  class Snmp4jAgentHBCtrlEntryRowFactory extends DefaultMOMutableRow2PCFactory
{
   public synchronized MOTableRow createRow(OID index, Variable[] values) throws
        UnsupportedOperationException {
      Snmp4jAgentHBCtrlEntryRow row = new Snmp4jAgentHBCtrlEntryRow(index,
          values);
      //--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::createRow
      row.setSnmp4jAgentHBCtrlLastChange(sysUpTime.get());
      row.setSnmp4jAgentHBCtrlEvents(new Counter64(0));
      //--AgentGen END
      return row;
    }

    public synchronized void freeRow(MOTableRow row) {
      //--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::freeRow
      //--AgentGen END
    }

    //--AgentGen BEGIN=snmp4jAgentHBCtrlEntry::RowFactory
    //--AgentGen END
  }

//--AgentGen BEGIN=_METHODS
  public void rowStatusChanged(RowStatusEvent event) {
    if (event.isDeniable()) {
      if (event.isRowActivated()) {
        // check column interdependent consistency
        Snmp4jAgentHBCtrlEntryRow row =
            (Snmp4jAgentHBCtrlEntryRow) event.getRow();
        if ((row.getSnmp4jAgentHBCtrlDelay().getValue() == 0) &&
            ((row.getSnmp4jAgentHBCtrlStartTime() == null) ||
             (DateAndTime.makeCalendar(
                 row.getSnmp4jAgentHBCtrlStartTime()).getTimeInMillis()
              <= System.currentTimeMillis()))) {
          event.setDenyReason(PDU.inconsistentValue);
        }
      }
    }
    else if (event.isRowActivated()) {
      Snmp4jAgentHBCtrlEntryRow row =
          (Snmp4jAgentHBCtrlEntryRow) event.getRow();
      HeartbeatTask task = new HeartbeatTask(row);
      if (row.getSnmp4jAgentHBCtrlDelay().getValue() == 0) {
        long startTime = DateAndTime.makeCalendar(
                 row.getSnmp4jAgentHBCtrlStartTime()).getTimeInMillis() -
            heartbeatOffset;
        heartbeatTimer.schedule(task,
                                new Date(startTime),
```

By using a row factory the columns of a row can be easily initialized. This can be particularly useful for rows that are restored from persistent storage. The example is here incomplete, because it always sets the last change timestamp, even it is provided by the values array. The latter is the case when the row is restored from persistent storage. In that case, the last change timestamp should not have been modified.

Most tables that use the `RowStatus` textual convention, need to trigger some actions when a row is activated and deactivated. A row is activated by setting its status to active(1) from another state than active(1). It is deactivated by setting its state to notInService(2) or destroy(6) from the active(1) state.
To react on row status changes, implement the `RowStatusListener` interface and register it at the `RowStatus` column of the corresponding table.
From the event object you can deduce whether the row status change is in the preparation phase (then it is deniable) or in the commit phase or undo phase (then it is not deniable).

```
                                    row.getSnmp4jAgentHBCtrlPeriod().getValue());
        }
        else {
          heartbeatTimer.schedule(task,
                                    row.getSnmp4jAgentHBCtrlDelay().getValue(),
                                    row.getSnmp4jAgentHBCtrlPeriod().getValue());
        }
        row.setUserObject(task);
      }
      else if (event.isRowDeactivated()) {
        Snmp4jAgentHBCtrlEntryRow row =
            (Snmp4jAgentHBCtrlEntryRow) event.getRow();
        HeartbeatTask task = (HeartbeatTask) row.getUserObject();
        if (task != null) {
          task.cancel();
        }
      }
    }
```

Sometimes managed objects need to be informed about row changes. Such a notification can be easily forwarded to an object implementing the `MOTableRowListener` interface. In this case the last change column gets updated with the current `sysUpTime` value whenever the row is changed.

```
    public void rowChanged(MOTableRowEvent event) {
      if (event.getRow() != null) {
        Snmp4jAgentHBCtrlEntryRow row =
            (Snmp4jAgentHBCtrlEntryRow) event.getRow();
        if (row.getSnmp4jAgentHBCtrlLastChange() != null) {
          row.getSnmp4jAgentHBCtrlLastChange().setValue(sysUpTime.get().getVal-
ue());
        }
      }
    }

    //--AgentGen END

  //--AgentGen BEGIN=_CLASSES

    class HeartbeatTask extends TimerTask {

      private Snmp4jAgentHBCtrlEntryRow configRow;

      public HeartbeatTask(Snmp4jAgentHBCtrlEntryRow configRow) {
        this.configRow = configRow;
      }

      public void run() {
        if (configRow.getSnmp4jAgentHBCtrlRowStatus().getValue() ==
            RowStatus.active) {
          long maxEvents = configRow.getSnmp4jAgentHBCtrlMaxEvents().getValue();
          if ((maxEvents > 0) &&
              (configRow.getSnmp4jAgentHBCtrlEvents().getValue() < maxEvents)) {
            configRow.getSnmp4jAgentHBCtrlEvents().increment();
            OID instanceOID =
                ((DefaultMOTable) snmp4jAgentHBCtrlEntry).
                getCellOID(configRow.getIndex(),
```

The `HeartbeatTask` class illustrates how notifications can be sent from instrumentation code. The task's `run` method is called by the `heartbeatTimer` for the scheduled points in time. A task is scheduled when the corresponding row status is activated and a task is canceled when the row is deactivated or destroyed.

```
                        idxSnmp4jAgentHBCtrlEvents);
          VariableBinding eventVB = new VariableBinding(instanceOID,
              configRow.getSnmp4jAgentHBCtrlEvents());
          snmp4jAgentHBEvent(notificationOriginator, context,
                             new VariableBinding[] {eventVB});
        }
        else {
          cancel();
          configRow.getSnmp4jAgentHBCtrlRowStatus().setValue(RowStatus.notIn-
Service);
        }
      }
      else {
        cancel();
      }
    }
  }

//--AgentGen END

//--AgentGen BEGIN=_END
//--AgentGen END
}
```

6 Advanced Concepts

The instrumentation concepts depicted so far are generally applicable to systems where the SNMP objects are individually coupled with the corresponding managed objects of the system.

If a SNMP interface is applied to a system that has already a management interface then additional precautions need to be taken to avoid redundant and unnecessary manual coding.

This section provides an introduction into concepts for generic instrumentation supported by SNMP4J-Agent.

6.1 Generic Instrumentation

With any sort of generic instrumentation, the pivotal question is how to associate an SNMP object instance with the corresponding object or value of the generic management interface. Often, the management interface is predetermined and cannot be changed when the SNMP interface is added.

The definition of the SNMP interface is a set of SMI MIB modules.

More often, the SNMP interface can be designed to simplify the object mapping - in some cases even the SNMP interface can be generated from an existing management interface specification.

Independently from the source of the MIB specification for the SNMP interface, a generic SNMP4J-Agent instrumentation needs to know which object/value of the generic instrumentation interface is represented by which SNMP object identifier.

If this mapping can be centralized, a generic instrumentation interface can be established as illustrated by Figure 7.

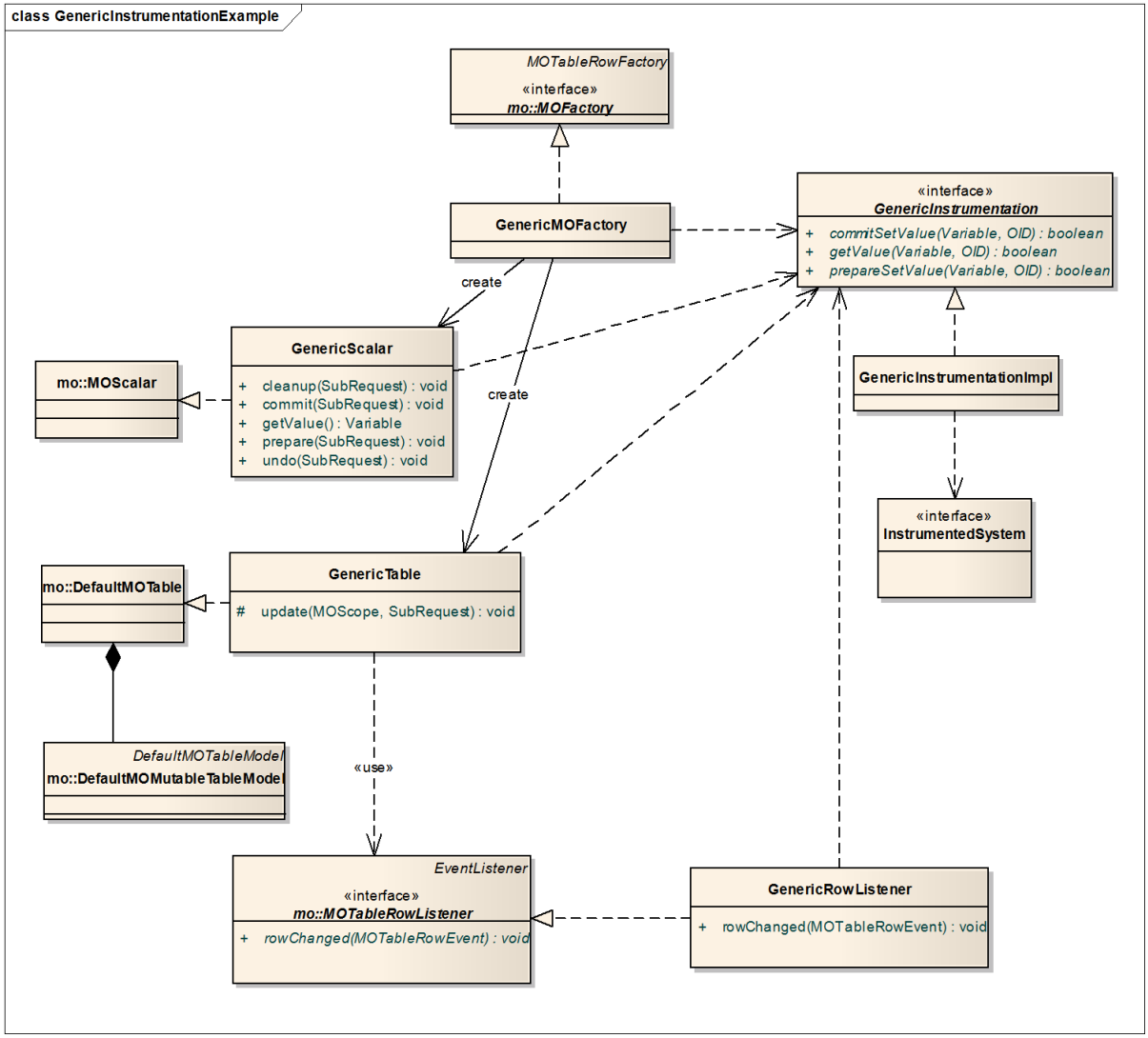

Figure 7: Generic Instrumentation Based on Custom MOFactory.

Following this instrumentation scheme, you will have to implement all `Generic*` classes and interfaces. The `GenericMOFactory` is then responsible to create your `MOScalar` and `DefaultMOTable` derivates with injected dependency to your `GenericInstrumentation`.

6.2 Agent Configuration and Fine Tuning

SNMP4J-Agent provides some advanced concepts to configure and fine tune an agent. Using these concepts is not required but helps to ensure best performance and quality (security).

6.2.1 Configuration

There are two configuration aspects for SNMP4J-Agent: *runtime* and *compile-time* configuration.

Compile-Time Configuration

A SNMP4J-Agent is typically built by the following components which need to be integrated:

- The `MOServer`(s) which manages access to the `ManagedObject`s.
- The `MessageDispatcher` which processes incoming and outgoing SNMP messages using a set of `TransportMapping`s.
- The `CommandProcessor` which processes `SnmpRequest`s on `ManagedObject`s.
- The `WorkerPool` which provides concurrency services for the `CommandProcessor` to be able to process messages concurrently independent from a special type of Java Runtime Environment.
- SNMPv3 message processing model boot counter and engine ID.
- The View Access Control Model (`VACM`) which controls access to MIB data.
- The User Security Model (`USM`) which manages SNMPv3 authentication and privacy.
- The `NotificationOriginator` which sends notifications (traps).
- The `SnmpTargetMIB` and the `SnmpCommunityMIB` MIB implementations needed to map SNMPv1 and v2c communities to SNMPv3 security names for integration into the VACM as well as the `SnmpNotificationMIB` needed to send and filter SNMP notifications.

The above components are easily integrated using an `AgentConfigManager` instance. In addition, the class provides means to read MIB data configuration (see "Runtime Configuration" on page 55) and read persistent MIB data. The result of the integration of these components is a running SNMP agent.

In most cases it is sufficient to provide the above components to the `AgentConfigManager` and then call its `run` method to start the agent. Internally, the `AgentConfigManager` integrates and configures the agent components in four phases:

1. Initialize

   - Creation of the `CommandProcessor` and its association with the supplied `WorkerPool`.
   - Creation of mandatory MIBs: SnmpTargetMIB, SNMPv2MIB, UsmMIB, VacmMIB, SnmpCommunityMIB, SnmpFrameworkMIB. Change events are connected between VACM, USM and the corresponding MIB implementations.
   - Creation of the `NotificationOriginator` and associating it to the `CommandProcessor`.
   - Creation of optional MIBs: `Snmp4jLogMIB` and `Snmp4jConfigMIB`.
   - Registration of user defined MIB modules at the `MOServer`(s) supplied to the `AgentConfigManager`.

2. Configure
   Loads configuration data from an `MOInput` source (if specified).

3. Restore State
   Restores a previously saved state of the MIB objects of the agent from a supplied `MOPersistenceProvider`. This operation is similar to the configuration phase, however it provides more control on how persistent data is restored and saved. The configuration, for example, always *creates and replaces* existing MIB data, whereas the default behavior of the persistent restore is *create and update*.

4. Launch
   Link `CommandResponder` and `MessageDispatcher`. By this link, the agent is ready to process SNMP requests. Then fire launch notifications (i.e., `coldStart` or `warmStart` notifications).

Each of the above phases can be run individually by directly calling it. The caller is responsible to comply with the above order. However, if you call the first two phases individually and then call `AgentConfigManager.run()`, it will take into account that the first two have already been called and that only the last two are left in order to run the agent.

The `AgentConfigManager` holds its initialization and configuration state internally in order to be able to incrementally configure the agent as well as to stop and restart it.

Runtime Configuration

Typically, the agent is configured at runtime regarding security settings, because those settings must not be hard coded and they must be available at agent start up. Of course, other settings can be configured at runtime as well, but the primary need for runtime configuration is motivated by security requirements.

SNMP4J-Agent provides a generic approach using Java properties to load MIB data at agent startup. This generic approach is not limited to security credentials. Any data for `ManagedObject`s supporting the `SerializedManagedObject` interface can be load and saved.

The properties format is favored over an XML format, because the overhead of XML processing is not acceptable for all application areas of SNMP4J-Agent.

The properties format is borrowed to a certain extend from the Structure of Management Information (SMI). The format distinguish between scalar and tabular data. Because MIB data is clustered by contexts since SNMPv3, a third configuration element named context is needed. The property format for *contexts* is:

The backslash „\" joins the current line with the next line to form a single property `key=value` *pair. With this special character you can specify a single property using multiple lines in a property file. Although this can significantly improve readability, you should always carefully check, that there is no blank or tab character after the backslash, because this will break the join.*

```
snmp4j.agent.cfg.contexts=\
{s|x}<ctx1>[,{s|x}<ctx2>...]
```

The format for *scalar* data is:

```
snmp4j.agent.cfg.oid[.ctx.<ctx>].<oid>=\
[{<format>}<value>]
```

The format for *tabular* data is:

```
snmp4j.agent.cfg.oid[.ctx.<ctx>].<oid>=\
[<numRows>:<numCols>]

snmp4j.agent.cfg.index[.ctx.<ctx>].oid.<rowIndex>=\
{o}<index>

snmp4j.agent.cfg.value[.ctx.<ctx>]\
.oid.<rowIndex>.<colIndex>=[{<format>}<value>]
```

The variable elements of the above formats are described by Table 3 on page 56.

VARIABLE	DESCRIPTION
`<ctx>`	The `<ctx>` variable is a SNMPv3 context name as UTF-8 string (format `{s}`) or a hexadecimal string (format `{x}`).
`<format>`	The `<format>` string is a choice of: ▸ `u` - an `Unsigned32` value. ▸ `i` - an `Integer32` value. ▸ `s` - an `OctetString` value. ▸ `x` - an `OctetString` value in hexadecimal format (separated by `:`). ▸ `d` - an `OctetString` value in decimal format (separated by `.`). ▸ `b` - an `OctetString` value in decimal format (separated by ' ' per byte). ▸ `n` - a Null value. ▸ `o` - an OID value as dotted string where string parts may be specified directly enclosed in single quotes (`'`) and an OID converted value of a variable/oid instance may be specified with the format `[#]{<name/oid>}`. The value of the variable will be included into the OID with prepended length if the `#` is used in the format string otherwise no length will be included. ▸ `t` - a TimeTicks value as an unsigned long value. ▸ `a` - a IpAddress value. ▸ `$` - gets the value from the variable or object instance specified by the name/oid following the `$`.
`<value>`	The variable value in the format specified by `format`.
`<numCols>`	The total number of columns in the table.

Table 4: The variables of the property format for managed object configuration.

VARIABLE	DESCRIPTION
`<numRows>`	The row index as a zero based unsigned integer.
`<rowIndex>`	The row index as a zero based unsigned integer which is *not* the SNMP row index.
`<colIndex>`	The column index as a zero based unsigned integer. which is *not* related to the column sub-index from the corresponding MIB specification
`<index>`	The OID value of the row's (SNMP) index.

Table 4: The variables of the property format for managed object configuration.

An example property file for configuring the system group of the SNMPv2-MIB is:

```
snmp4j.agent.cfg.contexts=
snmp4j.agent.cfg.oid.1.3.6.1.2.1.1.2.0={o}1.3.6.1.4.1.4976
snmp4j.agent.cfg.oid.1.3.6.1.2.1.1.4.0={s}System Administrator
snmp4j.agent.cfg.oid.1.3.6.1.2.1.1.6.0={s}<edit location>
snmp4j.agent.cfg.oid.1.3.6.1.2.1.1.7.0={i}10
snmp4j.agent.cfg.oid.1.3.6.1.2.1.1.9.1=0:2
```

The last line defines an empty table. It has the same effect as not providing the row at all.

Examples for tabular data configuration is provided below. Both tables refer to the `snmpEngineID` instance through the `$` and `o` formats. References can be used on hard coded as well as computed data which is available on agent start up or at least on `ManagedObject` creation. In addition, any `ManagedObject` configured before the current object (row) can be referenced. Using such references can improve configuration consistency.

```
## SNMP community MIB
snmp4j.agent.cfg.oid.1.3.6.1.6.3.18.1.1.1=1:7
snmp4j.agent.cfg.index.1.3.6.1.6.3.18.1.1.1.0={o}'public'
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.0={s}public
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.1={s}public
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.2={$1.3.6.1.6.3.10.2.1.1.0}
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.3={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.4={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.5={i}4
snmp4j.agent.cfg.value.1.3.6.1.6.3.18.1.1.1.0.6={i}1

## USM MIB
snmp4j.agent.cfg.oid.1.3.6.1.6.3.15.1.2.2.1=3:14
snmp4j.agent.cfg.index.1.3.6.1.6.3.15.1.2.2.1.0=\
{o}$#{1.3.6.1.6.3.10.2.1.1.0}.6.'SHADES'
```

```
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.0={s}SHADES
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.1={o}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.2={o}1.3.6.1.6.3.10.1.1.3
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.3={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.4={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.5={o}1.3.6.1.6.3.10.1.2.2
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.6={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.7={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.8={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.9={i}4
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.10={i}1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.11={s}SHADESAuthPassword
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.12={s}SHADESPrivPassword
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.0.13=
snmp4j.agent.cfg.index.1.3.6.1.6.3.15.1.2.2.1.1=\
{o}$#{1.3.6.1.6.3.10.2.1.1.0}.3.'SHA'
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.0={s}SHA
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.1={o}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.2={o}1.3.6.1.6.3.10.1.1.3
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.3={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.4={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.5={o}1.3.6.1.6.3.10.1.2.1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.6={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.7={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.8={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.9={i}4
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.10={i}1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.11={s}SHAAuthPassword
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.12=
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.1.13=
snmp4j.agent.cfg.index.1.3.6.1.6.3.15.1.2.2.1.2=\
{o}$#{1.3.6.1.6.3.10.2.1.1.0}.5.'unsec'
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.0={s}unsec
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.1={o}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.2={o}1.3.6.1.6.3.10.1.1.1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.3={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.4={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.5={o}1.3.6.1.6.3.10.1.2.1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.6={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.7={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.8={s}
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.9={i}4
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.10={i}1
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.11=
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.12=
snmp4j.agent.cfg.value.1.3.6.1.6.3.15.1.2.2.1.2.13=
```

The SNMP-USER-BASED-SM-MIB does not define a 12th column. SNMP4J-Agent uses those additional columns to persistently store pass-phrases using the `SerializedManagedObject` *interface. The property configuration can use those columns too, the specify the pass-phrases in clear text rather than by localized keys.*

6.2.2 Fine Tuning

There are several aspects of a SNMP4J agent which can be subject to fine tuning. The most noteworthy are *table size limits* and *thread pools*.

Table Size Limits

Since SNMP4J-Agent 1.4 table size limits can be simply applied on all tables in an agent by using the `DefaultMOServer` class or the even better the `AgentConfigManager`. The first class provides static methods to register a `MOTableRowListener` to all `MOTable` instances registered with a `MOServer`. The second class, `AgentConfigManager`, furthermore provides two methods to directly set table size limits:

- `setTableSizeLimit`
  The `setTableSizeLimit` method sets the specified limit to all tables in the agent. Its simple interface takes an integer value which represents the maximum number of rows for all tables.
- `setTableSizeLimits`
  The `setTableSizeLimits` method sets individual limits for the tables in the agent. It takes a `Properties` instance with key/value pairs where the key identifies a table object identifier or sub-tree. Thus, the size limit can be applied to a whole sub-tree at once or to individual tables.

Table size limits can help to prevent resource exhaustion by excessive row creation from external users.

Thread Pools

Thread pools (also called `WorkerPools`) are used to asynchronously process messages. Unless the are no idle threads in the pool, the caller will not block when starting a task to be executed by the pool. This characteristics is helpful when the instrumentation is not able to immediately respond all requests. More than four threads in the pool used by the `CommandProcessor` are only necessary, if there are potentially concurrent requests from several SNMP command generators and if instrumentation is slow (>0,1s).

To use any other thread pool between `TransportMapping` and `CommandResponder`, is generally not recommended. It causes additional overhead and does not improve concurrency.

For integration into Java EE environments, it can be useful to implement a `WorkerPool` subclass that uses concurrency services provided by the application server, instead using the `ThreadPool` provided by SNMP4J.

In any case, the pool is set either directly to the `CommandResponder` or by using the `AgentConfigManager` constructor.