Technical Standard

Application Response Measurement (ARM)

Issue 4.0 – Java Binding

THE Open GROUP

Technical Standard

Application Response Measurement (ARM) Issue 4.0 – Java Binding

ISBN: 1-931624-36-4

Document Number: C037

Published by The Open Group, October 2003.

Comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

Contents

1 Introduction 1
- 1.1 What is ARM? 1
- 1.2 How is ARM Used? 1
- 1.3 Selecting Transactions to Measure 2
- 1.4 The Evolution of ARM 3
- 1.5 Compatibility between ARM Versions 4
- 1.6 ARM 4.0 Java Bindings Overview 4
- 1.7 Terminology 4

2 Using the ARM 4.0 Java Binding 6
- 2.1 Example 6

3 Programming Options 8
- 3.1 ARM Measures Transactions 8
- 3.2 Application Measures Transactions 9
- 3.3 Selecting which Option to Use 9

4 Understanding the Relationships between Transactions 11
- 4.1 A Typical Distributed Transaction 11

5 Additional Data About a Transaction 14
- 5.1 Data Categories 14
  - 5.1.1 Counters 15
  - 5.1.2 Gauges 15
  - 5.1.3 Numeric IDs 15
  - 5.1.4 Strings 16
- 5.2 How to Provide the Additional Data 16
  - 5.2.1 Using ArmTransactionWithMetrics 16
  - 5.2.2 Using ArmTranReportWithMetrics 17
- 5.3 Processing Multiple Values of the Same Metric 17
  - 5.3.1 Counters 17
  - 5.3.2 Gauges 18
  - 5.3.3 Numeric IDs 18
  - 5.3.4 Strings 19

6 Creating ARM Objects 20
- 6.1 Overview of Java Interfaces 20
- 6.2 Creating ARM Objects in an Application 21
  - 6.2.1 Convenience Methods 23
- 6.3 Creating ARM Objects in an Applet 24

7 Error Handling Philosophy 26
7.1 Errors the Application Should Test For 26
7.2 Errors the Application Does Not Need to Test For........ 26
7.3 How an Application Tests for Errors 27
7.3.1 Testing a Method Return Code........ 28
7.3.2 Testing an Object Error Code 28
7.3.3 Registering a Callback........ 28

8 The ARM 4.0 Data Model 29
8.1 Data Model Using ArmTransaction........ 29
8.2 Data Model Using ArmTranReport 31

9 The org.opengroup.arm40.* Packages 32
9.1 Interface List (by Java Package) 32
9.2 Interface List (in Alphabetical Order)........ 33
9.3 Method Naming Conventions 34
9.4 org.opengroup.arm40.transaction.ArmApplication 35
9.5 org.opengroup.arm40.transaction.ArmApplicationDefinition........ 36
9.6 org.opengroup.arm40.tranreport.ArmApplicationRemote........ 37
9.7 org.opengroup.arm40.transaction.ArmConstants 37
9.8 org.opengroup.arm40.transaction.ArmCorrelator........ 38
9.9 org.opengroup.arm40.transaction.ArmErrorCallback 39
9.10 org.opengroup.arm40.transaction.ArmID........ 40
9.11 org.opengroup.arm40.transaction.ArmIdentityProperties 40
9.12 org.opengroup.arm40.transaction.ArmIdentityPropertiesTransaction..... 41
9.13 org.opengroup.arm40.transaction.ArmInterface........ 42
9.14 org.opengroup.arm40.metric.ArmMetric........ 42
9.15 org.opengroup.arm40.metric.ArmMetricCounter32........ 43
9.16 org.opengroup.arm40.metric.ArmMetricCounter32Definition 43
9.17 org.opengroup.arm40.metric.ArmMetricCounter64........ 43
9.18 org.opengroup.arm40.metric.ArmMetricCounter64Definition 44
9.19 org.opengroup.arm40.metric.ArmMetricCounterFloat32........ 44
9.20 org.opengroup.arm40.metric.ArmMetricCounterFloat32Definition 44
9.21 org.opengroup.arm40.metric.ArmMetricDefinition 44
9.22 org.opengroup.arm40.metric.ArmMetricFactory 45
9.23 org.opengroup.arm40.metric.ArmMetricGauge32 50
9.24 org.opengroup.arm40.metric.ArmMetricGauge32Definition........ 50
9.25 org.opengroup.arm40.metric.ArmMetricGauge64 50
9.26 org.opengroup.arm40.metric.ArmMetricGauge64Definition........ 50
9.27 org.opengroup.arm40.metric.ArmMetricGaugeFloat32........ 51
9.28 org.opengroup.arm40.metric.ArmMetricGaugeFloat32Definition.......... 51
9.29 org.opengroup.arm40.metric.ArmMetricGroup 51
9.30 org.opengroup.arm40.metric.ArmMetricGroupDefinition........ 52
9.31 org.opengroup.arm40.metric.ArmMetricNumericId32 52
9.32 org.opengroup.arm40.metric.ArmMetricNumericId32Definition........ 53
9.33 org.opengroup.arm40.metric.ArmMetricNumericId64 53
9.34 org.opengroup.arm40.metric.ArmMetricNumericId64Definition........ 53
9.35 org.opengroup.arm40.metric.ArmMetricString32........ 53

9.36 org.opengroup.arm40.metric.ArmMetricString32Definition.................. 54
9.37 org.opengroup.arm40.tranreport.ArmSystemAddress............................ 54
9.38 org.opengroup.arm40.transaction.ArmToken... 55
9.39 org.opengroup.arm40.tranreport.ArmTranReport 56
9.40 org.opengroup.arm40.tranreport.ArmTranReportFactory 59
9.41 org.opengroup.arm40.metric.ArmTranReportWithMetrics..................... 60
9.42 org.opengroup.arm40.transaction.ArmTransaction 61
9.43 org.opengroup.arm40.transaction.ArmTransactionDefinition................. 64
9.44 org.opengroup.arm40.transaction.ArmTransactionFactory..................... 65
9.45 org.opengroup.arm40.metric.ArmTransactionWithMetrics 67
9.46 org.opengroup.arm40.metric.ArmTransactionWithMetricsDefinition.... 68
9.47 org.opengroup.arm40.transaction.ArmUser 68

A Application Instrumentation Sample.. 70

B Information for Implementers.. 80

Preface

The Open Group

The Open Group, a vendor and technology-neutral consortium, has a vision of Boundaryless Information Flow achieved through global interoperability in a secure, reliable, and timely manner. The Open Group mission is to drive the creation of Boundaryless Information Flow by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate open specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

The Open Group provides opportunities to exchange information and shape the future of IT. The Open Group members include some of the largest and most influential organizations in the world. The flexible structure of The Open Group membership allows for almost any organization, no matter what their size, to join and have a voice in shaping the future of the IT world.

More information is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/testing.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/pubs.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Technical Standard for the Java Binding for Application Response Measurement (ARM) Issue 4.0. It has been developed and approved by The Open Group.

ARM is a standard for measuring service levels of single-system and distributed applications. ARM measures the availability and performance of transactions, both visible to the users of the business application and those visible only within the IT infrastructure.

Typographical Conventions

The following typographical conventions are used throughout this document:

- Arial font is used in text for Java elements.
- Arial Bold font is used in text for Java methods.
- Syntax and code examples are shown in fixed width font.

Trademarks

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this document:

- Mark Johnson, IBM
- Bill Furnas, Hewlett-Packard
- Marcus Thoss, tang-IT gmbH

Referenced Documents

The following documents are referenced in this document:

ARM 2.0	Technical Standard, July 1998, Systems Management: Application Response Measurement (ARM) (C807), published by The Open Group.
ARM 3.0	Technical Standard, October 2001, Application Response Measurement (ARM) Issue 3.0 Java Binding (C014), published by The Open Group.
ARM 4.0 (C Binding)	Technical Standard, October 2003, Application Response Measurement (ARM) Issue 4.0 – C Binding (C036), published by The Open Group.
IEEE Std 754-1985	IEEE Standard for Binary Floating-Point Arithmetic.
Unicode (UCS)	Refer to www.unicode.org.

1 Introduction

1.1 What is ARM?

It is hard to imagine conducting business around the globe without computer systems, networks, and software. People distribute and search for information, communicate with each other, and transact business. Computers are increasingly faster, smaller, and less expensive. Networks are increasingly faster, have more capacity, and are more reliable. Software has evolved to better exploit the technological advances and to meet demanding new requirements. The IT infrastructure has become more complex. We have become more dependent on the business applications built on this infrastructure because they offer more services and improved productivity.

No matter how much applications change, administrators and analysts responsible for the applications care about the same things they have always cared about:

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time experienced by the end-user?
- Which sub-transactions of the user transaction take too long?
- Where are the bottlenecks?
- How many of which transactions are being used?
- How can the application and environment be tuned to be more robust and perform better?

ARM helps answer these questions. ARM is a standard for measuring service levels of single-system and distributed applications. ARM measures the availability and performance of transactions (any units of work), both those visible to the users of the business application and those visible only within the IT infrastructure, such as client/server requests to a data server.

1.2 How is ARM Used?

ARM is a means through which business applications and management applications cooperate to measure the response time and status of transactions executed by the business applications.

Applications using ARM define transactions that are meaningful within the application. Typical examples are transactions initiated by a user and transactions with servers. As shown in Figure 1, applications on clients and/or servers call ARM when transactions start and/or stop. The agent in turn communicates with management applications, as shown in Figure 2, which provide analysis and reporting of the data.

The management agent collects the status and response time, and optionally other measurements associated with the transaction. The business application, in conjunction with the agent, may also provide information to correlate parent and child transactions. For example, a transaction that is invoked on a client may drive a transaction on an application server, which in turn drives ten other transactions on other application and/or data servers. The transaction on the client would be the parent of the transaction on the application server, which in turn would be the parent of the ten other transactions.

From the application developer's perspective ARM is a set of interfaces that the application loads and calls. What happens to the data after it calls the interfaces is not the developer's concern.

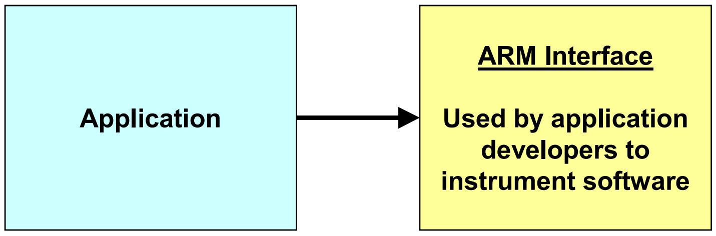

Figure 1: Application – ARM Interface

From the system administrator's perspective, ARM consists of the interfaces that applications load and call and the classes that implement these interfaces, plus programs to process the data, as shown in Figure 2. How the data is processed is not part of the ARM specification, but it is, of course, important to the system administrator.

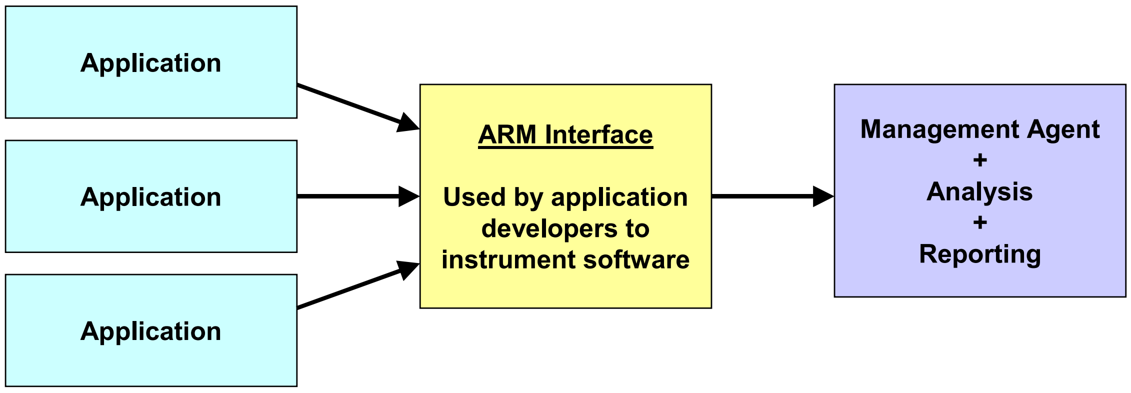
Application
Application
Application
ARM Interface
Used by application developers to instrument software
Management Agent + Analysis + Reporting

Figure 2: Application – ARM Management System Interaction

1.3 Selecting Transactions to Measure

ARM is designed to measure a unit of work, such as a business transaction, or a major component of a business transaction, that is performance-sensitive. These transactions should be something that needs to be measured, monitored, and for which corrective action can be taken if the performance is determined to be too slow.

Some questions to ask that aid in selecting which transactions to measure are:

- What unit of work does this transaction define?
- Are the transaction counts and/or response times important?
- Who will use this information?
- If performance of this transaction is too slow, what corrective actions will be taken?

1.4 The Evolution of ARM

ARM 1.0 was developed by Tivoli and Hewlett-Packard and released in June 1996. It provides a means to measure the response time and status of transactions. The interface is in the C programming language.

ARM 2.0 was developed by the ARM Working Group in 1997. The ARM Working Group was a consortium of vendors and end-users interested in promoting and advancing ARM. ARM 2.0 was approved as a Technical Standard of The Open Group in July 1998, part of the IT DialTone initiative. ARM 2.0 added the ability to correlate parent and child transactions, and to collect other measurements associated with the transactions, such as the number of records processed. The interface is in the C programming language.

ARM 3.0 was developed by The Open Group in 2001. It added new capabilities and specified interfaces in the Java programming language.

ARM 4.0 has been developed to implement new capabilities, and to provide equivalent functions for both C and Java programs. This document describes the Java program bindings. A companion document describes the C program bindings. ARM 4.0 adds the following capabilities:

- A richer and more flexible model for specifying application and transaction identity
- Tighter and more clarified semantics of registration
- Report attributes of a transaction that change on a per-instance basis
- Bind a transaction to a thread
- Indicate the amount of time a transaction is blocked waiting for an external event
- Indicate the true time when a transaction started executing for a specialized situation in which the standard indication of a start [*arm_start_transaction*() in ARM 4.0 C bindings] will not yield an accurate time
- Add a mechanism with which an application can register a callback that is called whenever an error occurs

1.5 Compatibility between ARM Versions

ARM defines low-level programming interfaces to be used between programs that are dynamically linked together. This limits how much an interface can change from version to version and still link with programs using a previous version. In particular, changing function entry points, or the call signatures of an entry point, prevents working with a different version.

ARM 1.0 and ARM 2.0 are interoperable. More specifically, any application instrumenting with ARM 1.0 can link to an ARM implementation using ARM 2.0, and *vice versa*. No other versions of ARM are similarly interoperable. It is expected that management agents may simultaneously support multiple versions of ARM. For example, a product may support both the ARM 3.0 and ARM 4.0 Java interfaces. However, a business application using the ARM 3.0 interfaces cannot instantiate the ARM 4.0 interfaces, and *vice versa*.

1.6 ARM 4.0 Java Bindings Overview

This specification describes three Java packages. Each package is equivalent to the block titled "ARM Interface" in Figure 1 and Figure 2.

- org.opengroup.arm40.transaction is the primary package that most applications use. The application calls a method when a transaction begins and ends, and the ARM implementation measures the response time.
- org.opengroup.arm40.tranreport is an alternative package that can be used by applications that measure the response time of their own transactions, and report the measurements after the fact.
- org.opengroup.arm40.metric can be used in addition to the org.opengroup.arm40.transaction package to report additional measurements about each transaction, such as a count of the amount of work accomplished.

An ARM implementation contains concrete classes that implement these interfaces. An ARM implementation may also be known as a "Management Agent" (and would be the part of the block labeled "Management Agent + Analysis + Reporting" that collects data). It is expected that companies will produce commercial ARM implementations, as was done for ARM 1.0 and ARM 2.0.

Business applications use ARM by creating objects that implement the interfaces in the packages, and then executing methods of the objects. The implementation of the classes takes care of all processing of the data, including moving the data outside the thread or JVM (Java Virtual Machine) process to be analyzed and reported.

1.7 Terminology

The following terminology is used throughout this document:

Can	Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that

	conforms to this document. An application can rely on the existence of the feature or behavior.
Implementation-defined	(Same meaning as "implementation-dependent".) Describes a value or behavior that is not defined by this document but is selected by an implementer. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations. The implementer shall document such a value or behavior so that it can be used correctly by an application.
May	Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. To avoid ambiguity, the opposite of "may" is expressed as "need not", instead of "may not".
Must	Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.
Shall	Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.
Should	For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. For an application, describes a feature or behavior that is recommended programming practice for optimum portability.
Undefined	Describes the nature of a value or behavior not defined by this document that results from use of an invalid program construct or invalid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.
Unspecified	Describes the nature of a value or behavior not specified by this document that results from use of a valid program construct or valid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.
Will	Same meaning as "shall"; "shall" is the preferred term.

2 Using the ARM 4.0 Java Binding

2.1 Example

An application uses ARM by creating objects that implement the ArmTransaction or ArmTranReport interfaces, and then invoking methods on these objects. A factory interface is provided to create the objects. The implementations of the ArmTransaction and ArmTranReport interfaces, which will generally be provided by software vendors, process the measurement data transparently to the application. Here is an example:

```
// This snippet assumes these are already created:
ArmTransactionFactory tranFactory;
BankAccount myAccount;

// These objects will be created at initialization:
ArmApplicationDefinition defnOnlineBank;
ArmApplication onlineBank;
ArmTransactionDefinition defnQueryBalance;
ArmTransaction queryBalance;

// Initialization: Create the application and transaction definitions:
defnOnlineBank = tranFactory.newArmApplicationDefinition("My Bank",
    null, null);
defnQueryBalance = tranFactory.newArmTransactionDefinition(
    defnOnlineBank, "QueryBalance", null, null);

// Initialization: Create the application and transaction objects:
onlineBank = tranFactory.newArmApplication(defnOnlineBank, null,
    null, null);
queryBalance = tranFactory.newArmTransaction(onlineBank,
    defnQueryBalance);

// At runtime: measure transactions as they execute:
queryBalance.start();
    // The following line is the real job of the application:
    status = myAccount.queryBalance(myCredentials);
queryBalance.stop(ARM_GOOD);
```

Many applications need no more. For a little more work, and a great deal more value, an application can link related transactions together. For example, a client transaction can be linked to an application server transaction, and the application server transaction can be linked to data server transactions. Understanding these relationships is tremendously useful for problem diagnosis, performance tuning, and capacity modeling. It also provides the means to link transactions to business transactions.

Here is an example of an application server, which both receives a correlation token from its parent, and passes a correlation token to its children. The additional code to handle the correlation tokens is highlighted in boldface type.

```
// This snippet assumes these are already created:
ArmTransactionFactory tranFactory;
BankAccount myAccount;

// These objects will be created at initialization:
ArmApplicationDefinition defnOnlineBank;
ArmApplication onlineBank;
ArmTransactionDefinition defnQueryBalance;
ArmTransaction queryBalance;

// Initialization: Create the application and transaction definitions:
defnOnlineBank = tranFactory.newArmApplicationDefinition("My Bank",
    null, null);
defnQueryBalance = tranFactory.newArmTransactionDefinition(
    defnOnlineBank, "QueryBalance", null, null);

// Initialization: Create the application and transaction objects:
onlineBank = tranFactory.newArmApplication(defnOnlineBank, null,
    null, null);
queryBalance = tranFactory.newArmTransaction(onlineBank,
    defnQueryBalance);

// At runtime: measure transactions as they execute:
byte[] parentBytes;  // Correlation token received from parent.
ArmCorrelator parent = tranFactory.newArmCorrelator(parentBytes);
queryBalance.start(parent);
ArmCorrelator corr = queryBalance.getCorr();
    // The following line is the real job of the application.
    // Note how the correlation token is sent along with the
    // transaction parameters.
    status = myAccount.queryBalance(myCredentials, corr);
queryBalance.stop(ARM_GOOD);
```

There are other optional features, but the examples above address many of the requirements for using ARM. One feature enables applications to provide additional information about a transaction, such as a count of the work done (for example, files processed).

3 Programming Options

The application has two options for providing measurement data:

- In Option 1, the preferred, and more widely used option, the application calls an ArmTransaction interface just before and after a transaction executes, and the ArmTransaction object makes the measurements.
- In Option 2, the application makes all the measurements itself and reports the data some time later. Option 2 should only be used in situations that preclude using Option 1.

3.1 ARM Measures Transactions

Figure 3 shows Option 1, the preferred, and most widely used option. The application creates an instance of ArmTransaction. Immediately prior to starting a transaction, the application invokes **start()**. The ArmTransaction instance captures and saves the timestamp. Immediately after the transaction ends, the application calls the **stop()** method, passing the status as an argument. The ArmTransaction instance captures the stop time. The difference between the stop time and the start time is the response time of the transaction. As soon as the **stop()** method returns, the application is free to reuse the ArmTransaction instance. The data will have already been copied from it to be processed.

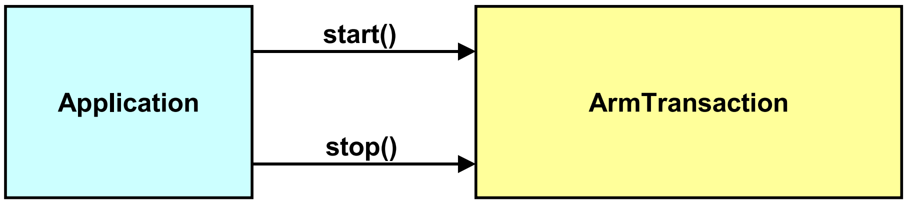

Figure 3: Measurement using Start/Stop

The application optionally provides any number of heartbeat and progress indicators using **update()** between a **start()** and a **stop()**. This is shown in Figure 4. Heartbeats are useful for long-running transactions.

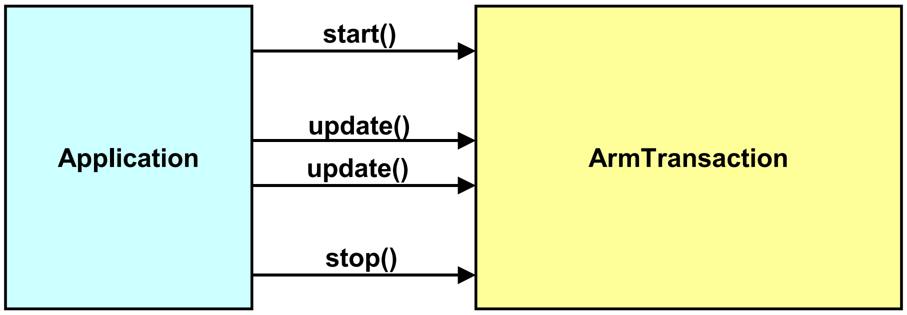

Figure 4: Application using Heartbeats

3.2 Application Measures Transactions

Figure 5 shows Option 2. The application itself measures the response time of the transaction. After the transaction completes (the delay could be short or long), it populates an ArmTranReport object with data, and calls **report()** to initiate processing of the data. As soon as the **report()** method returns, the application is free to reuse the ArmTranReport instance. The data will have already been copied from it to be processed.

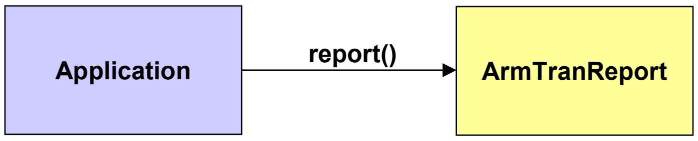

Figure 5: Measurement by the Application

3.3 Selecting which Option to Use

In many situations the business application can use either programming option. In general, the recommendation is to use Option 1 (ArmTransaction), unless that is not practical.

There is one situation for which the application must use Option 1:

- To provide heartbeats, the application must use the **update()** method of ArmTransaction between a **start()** and a **stop()**. Heartbeats are particularly valuable for long-running transactions. An ARM implementation may process updates, such as a real-time progress display, or check a threshold for a transaction that is taking too long.

There are two situations for which the application must use Option 2 (creating and populating ArmTranReport):

- Option 1 (ArmTransaction) requires that inline synchronous **start()** and **stop()** calls are used. The calls must be made at the moment the real transaction starts and stops. If they are not, the timings will not be accurate. If the application finds this inconvenient or impractical, the application must use Option 2 (ArmTranReport). ArmTranReport can be used because the application provides both the response time and the stop time.

- If the transaction executes on System A but is reported to ARM on System B, Option 2 must be used for all the reasons stated above. In addition, the application provides additional information that identifies the system and JVM (Java Virtual Machine) instance of the remote system where the transaction ran.

4 Understanding the Relationships between Transactions

There are several solutions available that measure transaction response times, such as measuring the response time as seen by a client, or measuring how long a method on an application server takes to complete. ARM can be used for this purpose as well. This is useful data, but it doesn't provide insight into how transactions on servers are related to business transactions executed by users or other application programs. ARM provides a facility for correlating transactions within and across systems. This section describes how this is done.

4.1 A Typical Distributed Transaction

Most modern applications consist of programs distributed across multiple systems, processes, and threads. Figure 6 is an example.

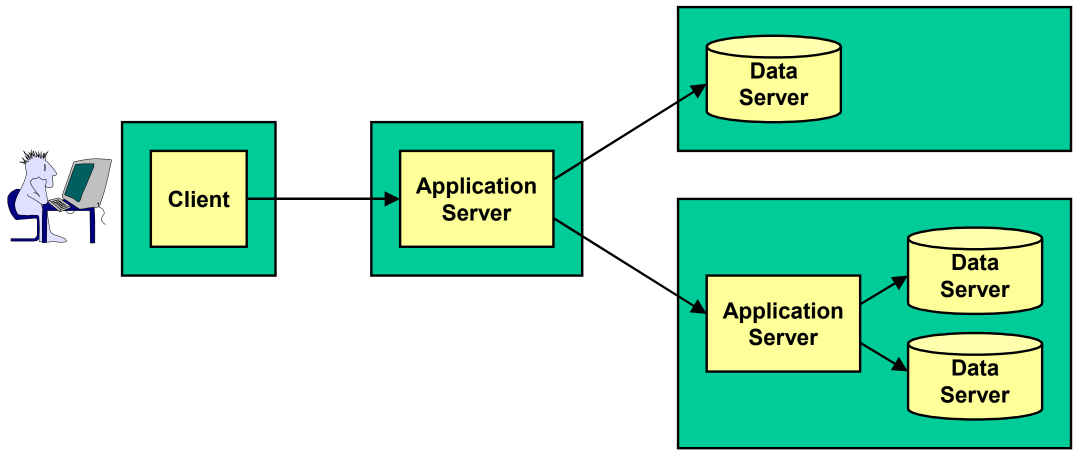

Figure 6: A Common Distributed Application Architecture

Figure 7 is an example transaction that runs on this application architecture. More correctly, Figure 7 shows a hierarchy of several transactions. To the user there is one transaction, but it is not unusual for the one transaction visible to the end-user to consist of tens or even over 100 sub-transactions.

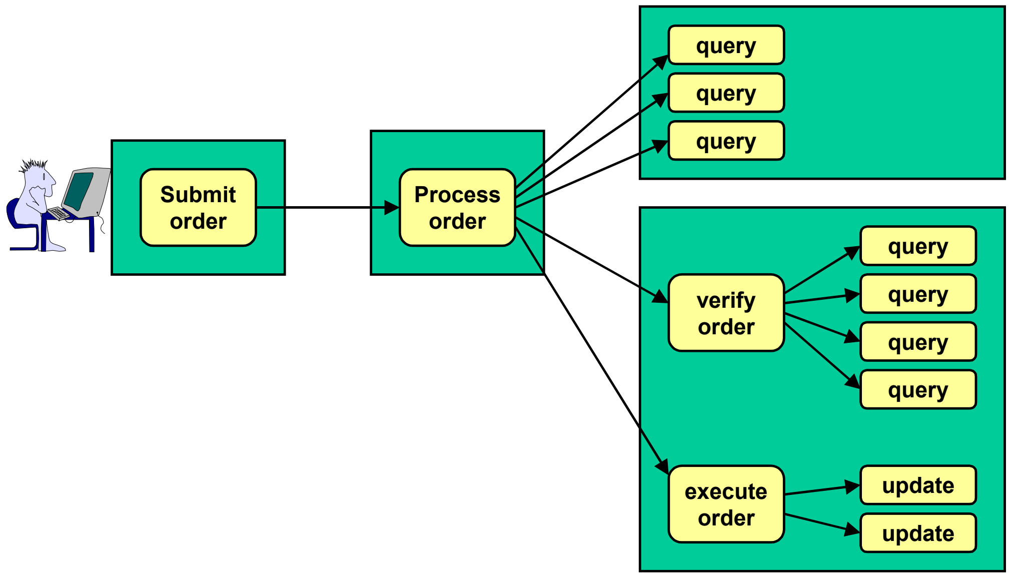

Figure 7: An Example of a Distributed Transaction

In ARM each transaction instance is assigned a unique token, named in ARM parlance a "correlator". To the application a correlator appears as an opaque byte array. There actually is a well-defined format to a correlator, and management agents and applications that understand it can take advantage of the information in it to determine where and when a transaction executed, which can aid enormously in problem diagnosis. Figure 8 shows the same transaction hierarchy as Figure 7, except that the descriptive names in Figure 7 have been replaced with identifiers. The lines are dotted instead of solid to indicate that without additional information, this would look to a management application like thirteen unrelated transactions.

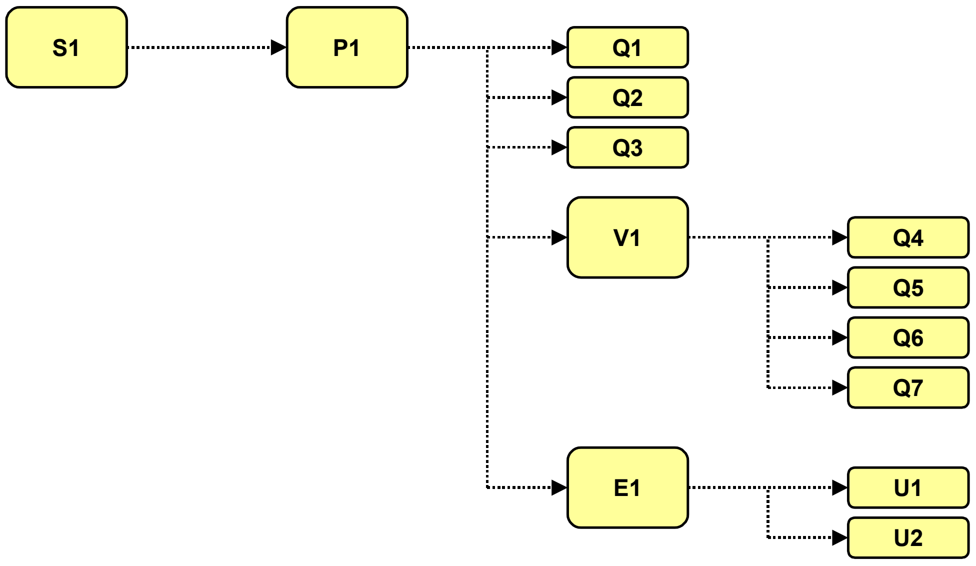

Figure 8: Distributed Transactions that Appear Unrelated

To relate the transactions together, the application components are each instrumented with ARM. In addition, each transaction passes the correlator that identifies itself to its children. In Figure 7 and Figure 8, the Submit Order transaction passes its correlator (S1) to its child, Process Order. Process Order passes its correlator (P1) to its five children – three queries, Verify Order, and Execute Order. Verify Order passes its correlator (V1) to its four children, and Execute Order passes its correlator (E1) to its two children.

The last piece in the puzzle is that each of the transactions instrumented with ARM passes its parent correlator to the ARM instrumentation class. The ARM instrumentation class knows the correlator of the current transaction. The correlators can be combined into a tuple of (parent correlator, correlator). Some of the tuples in Figure 8 are (S1,P1), (P1,Q1), (P1,E1), and (E1, U1). By putting the different tuples together, the management application can create the full calling hierarchy using the correlators to identify the transaction instances, as shown in Figure 9.

As an example of how this information could be used, if S1 failed, it would now be possible to determine that it failed because P1 failed, P1 failed because V1 failed, and V1 failed because Q6 failed.

Similar types of analysis could determine the source of response time problems. To analyze response time problems, additional information is needed. It's necessary to know if the child transactions execute serially, in parallel, or some combination of the two. The information may also be useful in locating unacceptable network latencies. For example, if the response time of S1 is substantially more than the response time of P1, and it is known that there is very little processing done on P1 that isn't accounted for in the measured response times, it suggests that there are unacceptable network or queuing delays between S1 and P1.

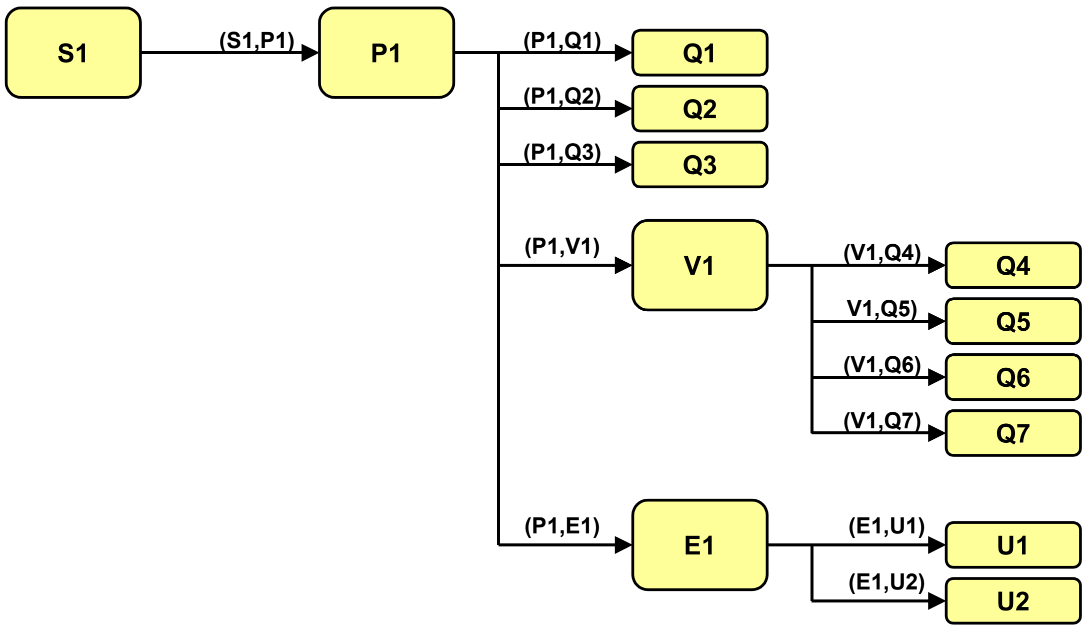

Figure 9: A Distributed Transaction Calling Hierarchy

5 Additional Data About a Transaction

The identification information and measurement information (status, response time, stop time) for any transaction measured with ARM provides a great deal of value, and there may be no requirement to augment the information. However, there are situations in which additional information could be useful, such as:

- How "big" is a transaction? Knowing a backup operation took 47 seconds may not be sufficient to know if the performance was good. Additional information, such as the number of bytes or files backed-up, provides much more meaning to the 47 seconds measurement.
- A transaction such as "get design drawings" may execute in less than a second for a simple part (e.g., a bracket). For complex parts, such as an engine, it may take many seconds to retrieve all the drawings, even if the system is performing well. Knowing the part number in this case makes the response time meaningful.
- The performance of a transaction will be affected by other workloads running on the same physical or logical system. Performance management tools may capture other information (e.g., CPU utilization) and combine it with response time measurements to plot the effect of CPU time on response time, which could be useful for planning the capacity of a system. However, other information that could be useful may not be available to performance management tools (e.g., the length of a queue internal to a program). It would be helpful for the application to provide this information.
- If a transaction fails it can be useful to know why. The required ARM status has four possible values: Good, Failed, Aborted, and Unknown. A detailed error code would be useful to understand why a transaction failed or was aborted. Capturing the code along with the other transaction information simplifies analysis by avoiding a later merge with, for example, error messages in a log file.

ARM provides a way for applications to provide these types of data. In ARM parlance the data is called "metrics". The use of metrics is *optional*.

ARM is not intended as a general-purpose interface for recording data. It is good practice to limit the use of metrics to data that is directly related to a transaction, and that helps to understand measurements about the transaction.

5.1 Data Categories

ARM supports nine data types. The data types are grouped in four categories. The categories are counters, gauges, numeric IDs, and strings.

5.1.1 Counters

A counter is a monotonically increasing non-negative value up to its maximum possible value, at which point it wraps around to zero and starts again. This is the IETF (Internet Engineering Task Force) definition of a counter.

A counter should be used when it makes sense to sum up the values over an interval. Examples are bytes printed and records written. The values can also be averaged, maximums and minimums (per transaction) can be calculated, and other kinds of statistical calculations can be performed.

ARM supports three counter types:

- 32-bit integer: ArmMetricCounter32
- 64-bit integer: ArmMetricCounter64
- 32-bit floating-point: ArmMetricCounterFloat32
  The floating-point standard is IEEE 754 (the same as the Java language).

5.1.2 Gauges

A gauge value can go up and down, and it can be positive or negative. This is the IETF definition of a gauge.

A gauge should be used instead of a counter when it is not meaningful to sum up the values over an interval. An example is the amount of memory used. If the amount of memory used over 20 transactions in an interval is measured and the average usage for each of these transactions was 15MB, it does not make sense to say that 20*15=300MB of memory were used over the interval. It would make sense to say that the average was 15MB, that the median was 12MB, and that the standard deviation was 8MB. The values can be averaged, maximums and minimums per transaction calculated, and other kinds of statistical calculations performed.

ARM supports three gauge types:

- 32-bit integer: ArmMetricGauge32
- 64-bit integer: ArmMetricGauge64
- 32-bit floating-point: ArmMetricGaugeFloat32
  The floating-point standard is IEEE 754 (the same as the Java language).

5.1.3 Numeric IDs

A numeric ID is a numeric value that is used as an identifier, and not as a measurement value. Examples are message numbers and error codes.

Numeric IDs are classified as non-calculable because it doesn't make sense to perform arithmetic with them. For example, the mean of the last seven message numbers would hardly ever provide useful information. By using a data type of numeric ID instead of a gauge or counter, the application indicates that arithmetic with the numbers is probably nonsensical. An agent could create statistical summaries based on these values, such as generating a frequency histogram by part number or error number.

ARM supports two numeric ID types:

- 32-bit integer: ArmMetricNumericId32
- 64-bit integer: ArmMetricNumericId64

5.1.4 Strings

A string is used in the same way that a numeric ID is used. It is an identifier, not a measurement value. Examples are part numbers, names, and messages.

The strings are in standard 16-bit Unicode (UCS-2) characters (the same as the Java language).

ARM supports one string type:

- Strings of 1-32 characters: ArmMetricString32

5.2 How to Provide the Additional Data

The application provides the values in one of two ways, depending on how the transaction data is measured.

5.2.1 Using ArmTransactionWithMetrics

If the application is calling ArmTransaction **start()** and **stop()**, it creates instances of subclasses of ArmMetric (e.g., ArmMetricCounter32) and binds an instance to an ArmTransactionWithMetrics instance using ArmMetricGroup. Each ArmMetric subclass supports the **set()** method. Figure 10 shows this process. ArmTransactionWithMetrics is a subclass of ArmTransaction and, hence, implements all the methods of ArmTransaction, in addition to some methods for manipulating metrics.

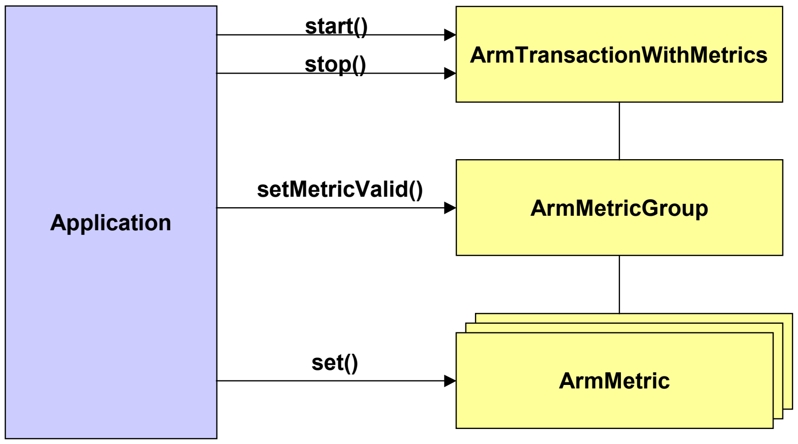

Figure 10: Providing Additional Data using ArmTransaction and ArmMetric

Prior to calling **start()**, **update()**, or **stop()**, the application may set the value in each metric. The ArmMetricGroup method **setMetricValid()** is used to indicate whether the data is valid. This is needed because the data might be valid only when **stop()** is executed, as an example. Figure 10 shows this process.

5.2.2 Using ArmTranReportWithMetrics

The process is similar if the application is calling ArmTranReport **report()**. It creates instances of subclasses of ArmMetric (e.g., ArmMetricCounter32) and binds an instance to an ArmTranReportWithMetrics instance using ArmMetricGroup. Each ArmMetric subclass supports the **set()** method. Figure 11 shows this process. ArmTranReportWithMetrics is a subclass of ArmTranReport and, hence, implements all the methods of ArmTranReport, in addition to some methods for manipulating metrics.

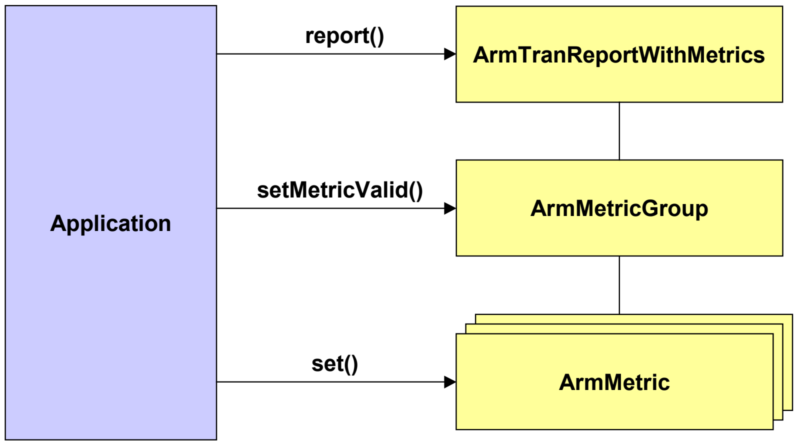

Figure 11: Providing Additional Data using ArmTranReport

Prior to calling **report()**, the application may set the value in each metric. The ArmMetricGroup method **setMetricValid()** is used to indicate whether the data is valid. This is needed because the data may not always be valid. Figure 11 shows this process.

5.3 Processing Multiple Values of the Same Metric

Additional semantics are defined when using ArmTransactionWithMetrics in order to eliminate ambiguity. The ambiguity arises because the metric may be valid on some or all of the **start()**, **update()**, and **stop()** method calls. The following sections describe the semantics for each of the data type categories.

5.3.1 Counters

If a counter is used, its initial value must be set at the time of the **start()** call. The difference between the value when the **start()** executes and when **stop()** executes (or the value in the last **update()** call if no metric value is passed in **stop()**) is the value attributed to this transaction. Similarly, the difference between successive **update()** calls, or from the **start()** to the first

update() call, or from the last **update()** to the **stop()** call, equals the value for the time period between the calls.

Here are three examples of how a counter would probably be used:

- The counter is set to zero at **start()** and to some value at **stop()** (or the last **update()** call). In this case, the application probably measured the value for this transaction and provided that value in the **stop()** call. The application always sets the value to zero at the **start()** call so the value at **stop()** reflects both the difference from the **start()** value and the absolute value.
- The counter is x1 at **start()**, x2 at its **stop()**, x2 at the next **start()**, and x3 at its **stop()**. In this case, the application is probably keeping a rolling counter. Perhaps this is a server application that counts the total workload. The application simply takes a snapshot of the counter at the start of a transaction and another snapshot at the end of the transaction. The agent determines the difference attributed to this transaction.
- The counter is x1 at **start()**, x2 at **stop()**, x3 (not equal to x2) at the next **start()**, and x4 at **stop()**. In this case, the application is probably keeping a rolling counter as in the previous example. But in this case the measurement represents a value affected by other users or transaction classes, so the value often changes from one **stop()** to the next **start()** for the same transaction class.

5.3.2 Gauges

Gauges can be set before **start()**, **update()**, and **stop()** calls. This creates the potential for different interpretations. If several values are provided for a transaction [one at **start()**, one at **update()**(s), and one at **stop()**], which one(s) should be used? In order to have consistent interpretation, the following conventions apply. Measurement agents are free to process the data in any way within these guidelines.

- The maximum value for a transaction will be the largest valid value passed at any time between and including the **start()** and **stop()** calls.
- The minimum value for a transaction will be the smallest valid value passed at any time between and including the **start()** and **stop()** calls.
- The mean value for a transaction will be the mean of all valid values passed at any time between and including the **start()** and **stop()** calls. All valid values will be weighted equally each time a **start()**, **update()**, or **stop()** executes.
- The median value for a transaction will be the median of all valid values passed at any time during the transaction. All valid values will be weighted equally each time a **start()**, **update()**, or **stop()** executes.
- The last value for a transaction will be the last valid value passed whenever any **start()**, **update()**, or **stop()** executes.

5.3.3 Numeric IDs

The last value passed when any of the **start()**, **update()**, or **stop()** calls are made will be the value attributed to the transaction instance. For example, if a value is valid at **start()** but not when any **update()** or **stop()** call executes, the value passed at the **start()** is used. If a value is valid when **start()** executes and when **stop()** executes, the value when **stop()** executes is the value for the transaction instance. This convention is identical to the string convention.

5.3.4 Strings

The last value passed when any of the **start()**, **update()**, or **stop()** calls are made will be the value attributed to the transaction instance. For example, if a value is valid at **start()** but not when any **update()** or **stop()** call executes, the value passed at the **start()** is used. If a value is valid when **start()** executes and when **stop()** executes, the value when **stop()** executes is the value for the transaction instance. This convention is identical to the numeric ID convention.

6 Creating ARM Objects

6.1 Overview of Java Interfaces

This document defines Java interfaces. A Java interface is an abstract specification of method signatures. Following is an example of an interface. This interface is named ArmMetricGroup; it is part of the org.opengroup.arm40.metric package, and it defines four method signatures: **getDefinition()**, **getMetric(**int**)**, **isMetricValid(**int**)**, and **setMetricValid(**int,boolean**)**.

```
package org.opengroup.arm40.metric;
public interface ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    public ArmMetric getMetric(int index);
    public boolean isMetricValid(int index);
    public int setMetricValid(int index, boolean value);
}
```

A program cannot create an instance (object) of an interface because there's no code to execute. Instead, a program creates an instance of a concrete class that *implements* the interface. In the following two code fragments, each of which would be in its own file, two classes are defined (MyGroup and AnotherVendorsOne). Each class declares that it implements the ArmMetricGroup interface (and they import all the class and interface definitions in the org.opengroup.arm40.metric so the Java compiler can reconcile all the names). Each class includes method bodies for at least the three methods defined in ArmMetricGroup. If it doesn't, the Java compiler will generate an error. Other methods may also be included. In these examples, MyGroup has one other method [**privateStuff()**] and AnotherVendorsOne has two other methods [**differentStuff1()** and **differentStuff2()**].

```
import org.opengroup.arm40.metric.*;
public class MyGroup
    implements ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    { // program code goes here }
    public ArmMetric getMetric(int index);
    { // program code goes here }
    public boolean isMetricValid(int index);
    { // program code goes here }
    public int setMetricValid(int index, boolean value);
    { // program code goes here }
    public int privateStuff();
    { // program code goes here }
}

import org.opengroup.arm40.metric.*;
public class AnotherVendorsOne
```

```
    implements ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    { // program code goes here }
    public ArmMetric getMetric(int index);
    { // program code goes here }
    public boolean isMetricValid(int index);
    { // program code goes here }
    public int setMetricValid(int index, boolean value);
    { // program code goes here }
    private void differentStuff1()
    { // program code goes here }
    private void differentStuff2()
    { // program code goes here }
}
```

To create an object that implements the ArmMetricGroup interface, a program could create an instance of either MyGroup or AnotherVendorsOne. By assigning the object to a variable of type ArmMetricGroup, this variable can be used as if ArmMetricGroup is a concrete class. In the following code snippet, group is of type MyGroup and can execute any of the five methods of MyGroup. g5 is of type ArmMetricGroup so it can execute only the four methods in ArmMetricGroup. The program statement **g5.privateStuff()** would generate a compiler error because **privateStuff()** is not defined in the ArmMetricGroup interface, whereas **group.privateStuff()** does not result in a compiler error.

```
MyGroup group = new MyGroup();
ArmMetricGroup g5 = (ArmMetricGroup) group;
int mySlot = 4;
g5.setMetricValid(mySlot, true);
group.setMetricValid(mySlot, false);
group.privateStuff();
```

6.2 Creating ARM Objects in an Application

The discussion so far has been a short tutorial on Java interfaces and would apply to any Java program. The remainder of this section describes how applications (not applets) create objects that implement the ARM 4.0 Java Bindings interfaces. The next section describes how applets create the objects.

A fundamental characteristic of ARM is that an application that uses ARM will be able to work with any ARM implementation, whether written in-house or purchased from a vendor. Vendors compete with each other to provide better ARM implementations. The use of Java interfaces creates a potential problem because each vendor will have its own names for its own classes. In the examples above, MyGroup and AnotherVendorsOne are names that are not part of the ARM specification. Further, a program that uses ARM should never use either name in a program because if it does, that program is restricted to only working with the ARM implementation from that particular vendor. But a program cannot create an instance of an object which implements an interface, such as the ArmMetricGroup interface, without naming a specific class. It is a compiler error to code:

```
ArmMetricGroup g5 = new ArmMetricGroup();
```

So how can a program create a concrete class without naming the class directly?

ARM uses two mechanisms to create objects that implement the ARM interfaces. Together, these mechanisms permit a system administrator to choose an ARM implementation regardless of the class names of the implementation while allowing the application to work with any ARM implementation.

1. This document defines three factory interfaces, one for each package. New objects are created by first creating an object that implements a factory interface, then invoking methods of the factory interface. The factory methods are used instead of using the Java new operator. The three factory interfaces are ArmMetricFactory, ArmTranReportFactory, and ArmTransactionFactory.

2. Using factory interfaces alone does not avoid naming the classes in each ARM implementation, because the objects implementing the factory interfaces need to be created by name. The application does not know the factory class names in advance (otherwise it would only work with one ARM implementation). The application gets the names of the factory classes through the use of the Java system properties. A system administrator assigns the names of the factory classes to the properties before starting an application that uses ARM.

The remainder of this section describes the process in more detail.

All JDKs implement the java.lang.System and java.util.Properties classes. These classes contain several methods to manipulate properties. java.util.Properties is a hash table containing properties. A property is a string and it is referenced within the hash table by a key, which is also a string. Java programs can create instances of java.util.Properties for their own purposes. java.lang.System creates a special instance of java.util.Properties that is a singleton within the JVM (Java Virtual Machine). It provides a single place to store property values that will be available to all programs running within the JVM.

ARM defines three property keys, one for each of the three factory classes. Each factory interface defines a static (class) constant named propertyKey. The value of each constant is the same name as the factory interface. For example, the ArmMetricFactory interface assigns the value ArmMetricFactory to its constant variable propertyKey. Each of the following statements is in the respective factory interfaces:

```
public static final String propertyKey = "Arm40.ArmMetricFactory";
public static final String propertyKey = "Arm40.ArmTranReportFactory";
public static final String propertyKey = "Arm40.ArmTransactionFactory";
```

During initialization of the ARM environment in a JVM, a program provided by the system administrator will assign a class name in the system properties for each property key. For example, the following code snippet assigns the class name com.vendor1.arm.ArmTranFactory to the property key for ArmTransactionFactory. This would be repeated for the other three factory interfaces. In this case a company with a domain name of vendor1.com presumably supplies the ARM implementation.

```
Properties p = System.getProperties();
String valueTranFactoryClass = "com.vendor1.arm.ArmTranFactory";
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
p.put(keyTranFactoryClass, valueTranFactoryClass);
```

To create any of the ARM objects an application first creates an instance of the appropriate factory class. It then uses the methods of the factory class to create the objects that implement the ARM interfaces. It is common for an application to create one instance of each of the four factory classes during initialization, and then use them to create all the other objects. However, there is no requirement to do so – the application can create any number of instances of each factory, and can create one whenever it needs one.

In the following code snippet, tranFactoryName is the name of the factory class (for example, com.vendor1.arm.ArmTranFactory), tranFactoryClass is the factory class (all Java classes can be represented by an instance of java.lang.Class), and tranFactory is an instance of the factory class. tranFactory can be used to create any number of instances of ArmTransaction and related classes. Three are created in this example, all for the same transaction ID. In this example, the **Class.forName()** method specifically names the system class loader, which can avoid problems if multiple class loaders are involved and the factory implementation uses JNI (Java Native Interface).

```
// Create the factory object
Properties p = System.getProperties();
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
String tranFactoryName = p.getProperty(keyTranFactoryClass);
Class  tranFactoryClass = Class.forName(tranFactoryName, true,
    ClassLoader.getSystemClassLoader());
ArmTransactionFactory tranFactory;
tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();

// The application and transaction objects
ArmApplicationDefinition myAppDefn;
ArmApplication myApp;
ArmTransactionDefinition myTranDefn;
ArmTransaction tran1, tran2, tran3;

// Create application objects
myAppDefn = tranFactory.newArmApplicationDefinition("My Application",
    null, null);
myApp = tranFactory.newArmApplication(myAppDefn, null, null, null);

// Create three transaction objects
myTranDefn = tranFactory.newArmTransactionDefinition(
    myAppDefn, "My Transaction", null, null);
tran1 = tranFactory.newArmTransaction(myApp, myTranDefn);
tran2 = tranFactory.newArmTransaction(myApp, myTranDefn);
tran3 = tranFactory.newArmTransaction(myApp, myTranDefn);
```

6.2.1 Convenience Methods

ARM implementations or SDKs may (but are not required to) provide convenience routines that hide some of these details. A suggested way to do this would be to provide class methods to create the factories by embedding the logic from above. For example:

```
public class ArmFactory
{
    public static ArmTransactionFactory createArmTransactionFactory(){}
    public static ArmTransReportFactory createArmTransReportFactory(){}
    public static ArmMetricFactory createArmMetricFactory(){}
}
```

6.3 Creating ARM Objects in an Applet

The approach of using the system properties to identify the names of the concrete classes works for Java applications but it will not work for Java applets. Applets do not have access to the system properties, except a few that are expressly permitted. This section describes how to provide the class names to applets.

The basic principles remain the same. The applet should not change even if the ARM implementation changes. A system administrator should control which ARM implementation is used by each applet. This will be the system administrator of the server from which the applet is loaded.

The Java language permits applets to access files on the server system from which they originated, as long as those files are in the applet's codebase. By default the codebase is the directory that contains the HTML file that loaded the applet. The HTML file can specify the codebase to be a different directory using the CODEBASE tag. Classes in the applet's unnamed package (any class that doesn't specify a package) are taken from the same directory as the codebase. For classes that are members of a package, the codebase is extended by the package name. For example, if the codebase is www.abc.com/test, the ARM package would be in directory www.abc.com/test/org/opengroup/arm40/transaction. An applet can get the URL of the codebase using the **getCodeBase()** method of the java.applet class.

ARM defines a similar mechanism for applets as for applications. The main difference is that an application gets its properties from the system properties, whereas an applet gets its properties from a file named arm.properties. arm.properties must reside in the codebase of the applet. An application uses **java.lang.System.getProperties()** to create an instance of the java.util.Properties class. An applet creates an instance of java.util.Properties by loading it from the data in arm.properties. The format of each property, and the keys that identify the three factory classes, are identical.

The following example shows how an applet would initialize its Properties object and create an instance of ArmTransactionFactory, then use the factory to create an instance of ArmTransaction.

```
// Retrieve properties file from codebase
URL urlCodeBase = getCodeBase();
URL urlArmProp = new URL(urlCodeBase.getProtocol(),
    urlCodeBase.getHost(),
    urlCodeBase.getPort(),
    urlCodeBase.getFile()
    +"arm.properties");
InputStream in = urlArmProp.openStream();
Properties armProp = new Properties();
armProp.load(in);

// Get the factory class name and create an instance
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
String tranFactoryName = armProp.getProperty(keyTranFactoryClass);
Class tranFactoryClass = Class.forName(tranFactoryName);
```

```
ArmTransactionFactory tranFactory;
tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();

// The application and transaction objects
ArmApplicationDefinition myAppDefn;
ArmApplication myApp;
ArmTransactionDefinition myTranDefn;
ArmTransaction tran1, tran2, tran3;

// Create application objects
myAppDefn = tranFactory.newArmApplicationDefinition("My Application",
    null, null);
myApp = tranFactory.newArmApplication(myAppDefn, null, null, null);

// Create transaction objects
myTranDefn = tranFactory.newArmTransactionDefinition(myAppDefn,
    "My Transaction", null, null);
tran1 = tranFactory.newArmTransaction(myApp, myTranDefn);
```

One final restriction, imposed by the Java language on applets, is that the implementation of the interface – that is, the concrete classes that implement the ARM interfaces – must also reside in the applet's codebase.

7 Error Handling Philosophy

The error handling philosophy of the ARM specification can be summed up as the following:

"Programmers and system administrators need to know about errors; programs do not."

The practical effect of this philosophy is that applications do not need to check for errors, except when creating factory classes, when exceptions could be thrown.

An application that contains programming errors, or that receives invalid data, could generate invalid measurement data. This is a problem that programmers and system administrators should correct. But at runtime there's nothing an application can do about it, so the ARM interface takes the approach of being as unobtrusive as possible, and permitting the application logic to flow normally. Programmers testing programs, and system administrators managing systems using ARM, should check for error reports from ARM implementations.

Any method that creates an ARM object, or a copy of an ARM object, will always return a valid non-null object of that type. If invalid data is provided, the data within that object may be incorrect or meaningless. However, the object will be syntactically correct; that is, it will be a valid Java object, and any of its methods can be invoked without causing an exception.

7.1 Errors the Application Should Test For

There is one situation in which the application needs to test for errors. This situation is when the application is creating factory classes. When doing so, there are three exceptions, all thrown by the static methods of java.lang.Class, which the application should be prepared to catch. They most likely indicate that the ARM environment has not been initialized correctly. For example, an ARM implementation may not be in the class path, or the java.lang.properties file may contain invalid class names. These exceptions are:

- java.lang.ClassNotFoundException signals that a class to be loaded could not be found. It is thrown by **Class.forName()**.
- java.lang.IllegalAccessException signals that a class or initializer is not accessible. It is thrown by **Class.newInstance()**.
- java.lang.InstantiationException signals an attempt to instantiate an interface or an abstract class. It is thrown by **Class.newInstance()**.

7.2 Errors the Application Does Not Need to Test For

None of the interfaces in the three packages that comprise the ARM specification define exceptions, and none throw exceptions, except those that are so pervasive in Java classes that they do not have to be declared. Exceptions that do not have to be declared are those that are subclasses of java.lang.Error or java.lang.RuntimeException. Such exceptions can be thrown by practically any method. One RuntimeException that is often encountered is ArrayIndexOutOfBoundsException. Examples of exceptions that do have to be declared are IOExceptions.

Here are some examples of errors in the use of the ARM interface that are not exceptions:

- The application passes a null pointer when a non-null pointer is required.
- The application passes an invalid format ID or status value.
- The application passes an incorrectly formed correlator (which it may have received from the program that called it, in which case the problem is in the program or system that called it).
- When using ArmTransaction, two **start()** methods are executed consecutively without an intervening **stop()** or **reset()**, or an **update()** method is executed without a **start()** first being executed.

ARM has two principles for handling this type of error:

- The programmer (during program development) and/or the system administrator (after the application has been deployed into production) need to be aware of them so the problem can be corrected.
- An application running in production does not need to be aware of and test for them, but has the opportunity to do so. To enable this, an application can test method return codes, get an error status from an object, and register a callback that is invoked when an error occurs.

The recommended approach is for the ARM implementation to have a mechanism for providing programmers and/or system administrators with error notification. For example, the ARM implementation could write the data to a log file and/or create and send an error event to an event console. During development and testing, the programmer would inspect the log file for errors. During production a system administrator would do the same. In this way a system administrator will have one place to look for errors for all applications using ARM.

Except for the callback via the ArmErrorCallback interface, the content, format, and delivery mechanisms for error notifications are not part of the ARM specification. They are implementation-defined. A good implementation will provide sufficient detail to not only detect that a problem occurred, but to also isolate and resolve the problem.

7.3 How an Application Tests for Errors

If an application chooses to test for errors that don't result in throwing an exception, it may do so in three ways. Note again that testing for these errors is entirely optional. They can be ignored (the data about the transactions and related entities may not be valid or useful to an ARM implementation, but the application's execution will not be disrupted in any way).

Anytime a method results in an error the ARM implementation does the following:

- Sets the object's error code. This error code can be retrieved with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive.
- Calls **errorCodeSet()** in ArmErrorCallback, if an ArmErrorCallback has been successfully registered with the ArmApplication.
- Returns the error code if the method that caused the error returns an error code.

Anytime a method of an object is invoked, any previous error code value is overwritten with the status of the last method invoked.

7.3.1 Testing a Method Return Code

Many methods return an int. Unless otherwise noted, the returned integer is an error code. It has a value of zero if no error occurred. Any non-zero value indicates that there was an error.

7.3.2 Testing an Object Error Code

Many methods may result in the ARM implementation detecting an error. Some of these methods return an error code and some do not. Regardless of whether a method returns an error code, all methods that detect an error set the object's error code. The application may retrieve the error code with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive. The application may use **getErrorMessage()** to get a string message describing the error.

7.3.3 Registering a Callback

An application may create an object that implements the ArmErrorCallback interface and register it with the **setErrorCallback()** method of ArmApplication. **setErrorCallback()** returns a boolean indicating whether the registration was accepted. If it was accepted, any method that causes an object's error code to be set to a non-zero value will cause the ArmErrorCallback method **errorCodeSet()** to be invoked, passing a reference to the object and the name of the interface and method in which the error was detected.

8 The ARM 4.0 Data Model

8.1 Data Model Using ArmTransaction

Figure 12 shows the data model when using ArmTransaction. To avoid clutter in the diagrams, some interfaces, such as ArmCorrelator, ArmID, and ArmUser, are not shown as separate classes.

Four interfaces must be used: ArmApplicationDefinition, ArmTransactionDefinition, ArmApplication, and ArmTransaction. All others are optional. The relationships between interfaces are implemented via Java object references.

- ArmApplicationDefinition describes metadata about the application. It is the root object that anchors all other objects. There is one per application. Note, however, that an ARM application is a logical entity. The same process may support several very different logical applications.

- ArmTransactionDefinition describes metadata about a transaction. There may be any number of transaction definitions in each application.

- ArmIdentityProperties and its subclass ArmIdentityPropertiesTransaction are used to provide additional optional metadata about applications and transactions.

- ArmApplication represents an application instance. It is common for there to be one instance per definition, but there could be any number of instances per definition.

- ArmTransaction represents an executing transaction. It is the most important and the most widely used ARM interface. More specifically, the application uses the paired methods **start()** and **stop()** to indicate the beginning and end of the transaction. At all other times the ArmTransaction object is unused and does not represent anything. ArmTransaction objects may be reused. An application may create one ArmTransaction per thread, or it might create a pool of ArmTransaction instances, and reuse them on an as-needed basis.

  The measurements common to all transaction instances are status, response time, and the time-of-day when the transaction executed. Optionally, the application can also provide data to correlate parent and child transactions and context data such as the user on whose behalf the transaction executes. Substitute ArmTransactionWithMetrics if additional metric data is provided.

- ArmTransactionWithMetrics is a subclass of ArmTransaction. If metrics are used (see the following descriptions of ArmMetric and ArmMetricGroup), ArmTransactionWithMetrics is used instead of ArmTransaction.

- ArmMetric (and its ten subclasses, such as ArmMetricCounter32). The application can optionally augment the basic and correlation data with other data, called “metrics”. A metric is a numeric or string value. It may represent a counter (such as the number of files processed), a gauge (such as the queue length when the transaction executes), or information (such as an error number or the name of a file that was processed). An application creates ArmMetric instances and associates them with one or more ArmTransactionWithMetrics instances. When ArmTransactionWithMetrics methods are processed, the values in the ArmMetric instances are captured and considered part of the data for the transaction.

A benefit of this approach is that the same ArmMetric object can be shared by many instances of the same transaction or different transactions. Updating the value in one place (the ArmMetric object) effectively propagates it to many ArmTransactionWithMetrics objects, though the data is only captured when a **start()**, **update()**, or **stop()** call is made.

- ArmMetricGroup is used to bind a set of ArmMetric objects to an ArmTransactionWithMetrics object.
- ArmTransactionWithMetricsDefinition, ArmMetricGroupDefinition, and ArmMetricDefinition are used to provide metadata about metric groups and metrics.

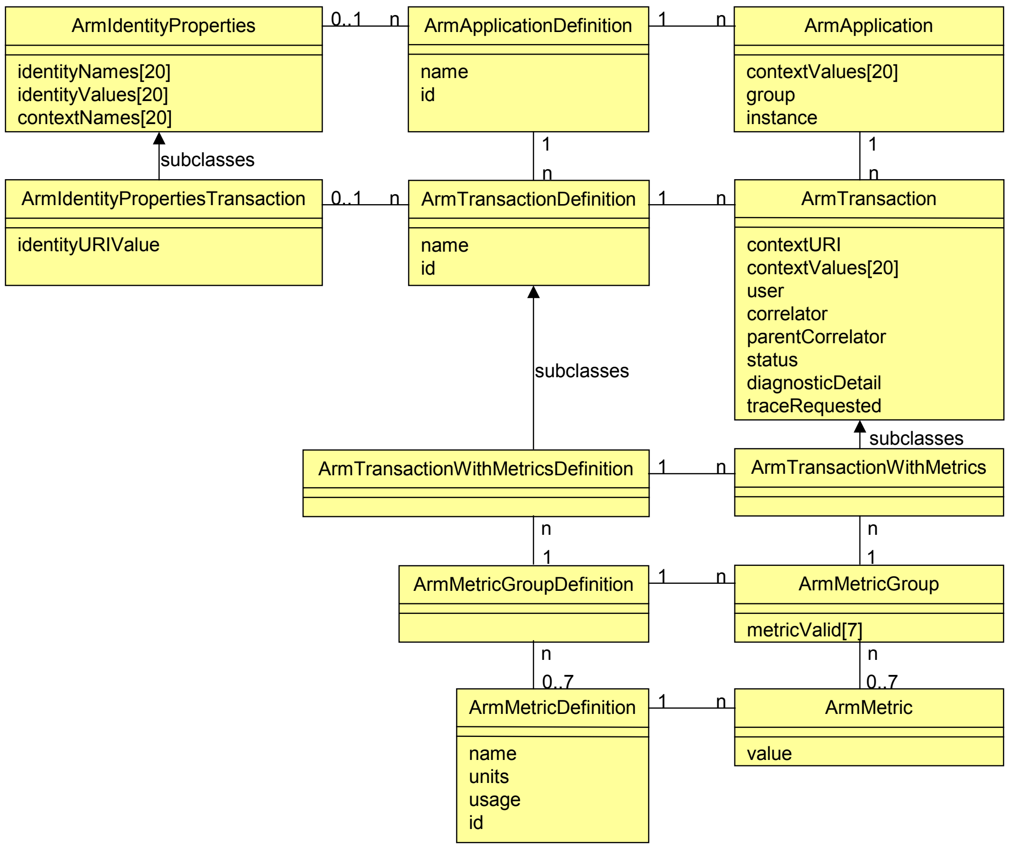

Figure 12: ARM 4.0 Data Model Using ArmTransaction

8.2 Data Model Using ArmTranReport

Figure 13 summarizes the data model when using ArmTranReport. All the interfaces that are already depicted in Figure 12 and described above are not shaded. Only the interfaces that specifically support the use of ArmTranReport are shared and described below.

- ArmTranReport and its subclass ArmTranReportWithMetrics contain the information about a completed transaction. Typically an application might create one ArmTranReport instance for each type of transaction that it executes, or a pool of them if it is multi-threaded. When a transaction completes, the application extracts one of the objects, populates it, then calls **report()**. As soon as **report()** returns, the application can reuse the ArmTranReport instance.
- ArmApplicationRemote may execute on the local system or a remote system. When it executes on the local system, it is associated with an ArmApplication object. When it executes on a remote system, it is associated with a subclass of ArmApplication – ArmApplicationRemote. ArmApplicationRemote differs from ArmApplication only in that it may be associated with an ArmSystemAddress object. An ArmSystemAddress object contains the network address of the remote system in one of several possible formats.

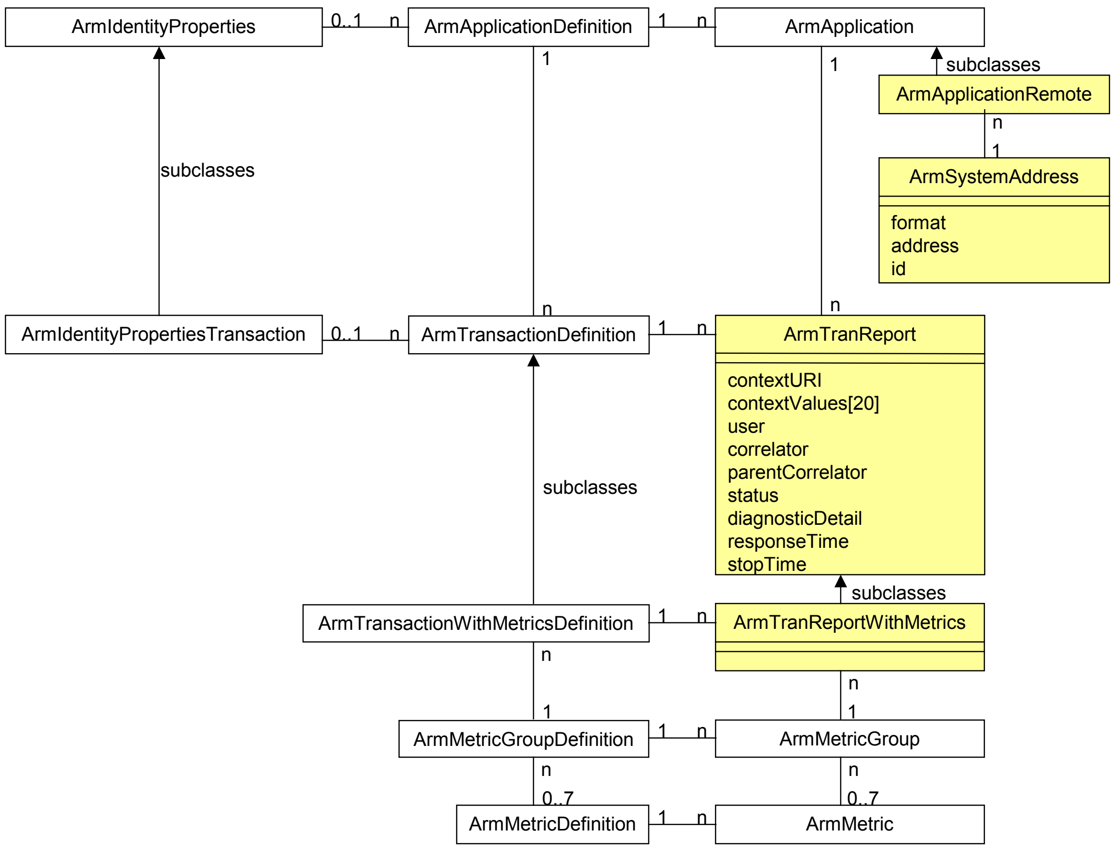

Figure 13: ARM 4.0 Data Model Using ArmTranReport

9 The org.opengroup.arm40.* Packages

9.1 Interface List (by Java Package)

The following table lists all the interfaces, arranged by package name, and in alphabetical order within the package. Most applications will use only the transaction package and can ignore the other packages, which address specialized requirements.

There is one factory interface for each package. Use this factory interface to create instances of the other interfaces.

org.opengroup.arm40. transaction	**org.opengroup.arm40. tranreport**	**org.opengroup.arm40.metric**
ArmApplication	ArmApplicationRemote	ArmMetric
ArmApplicationDefinition	ArmSystemAddress	ArmMetricCounter32
ArmConstants	ArmTranReport	ArmMetricCounter32Definition
ArmCorrelator	ArmTranReportFactory	ArmMetricCounter64
ArmErrorCallback		ArmMetricCounter64Definition
ArmID		ArmMetricCounterFloat32
ArmIdentityProperties		ArmMetricCounterFloat32Definition
ArmIdentityPropertiesTransaction		ArmMetricDefinition
ArmInterface		ArmMetricFactory
ArmToken		ArmMetricGauge32
ArmTransaction		ArmMetricGauge32Definition
ArmTransactionDefinition		ArmMetricGauge64
ArmTransactionFactory		ArmMetricGauge64Definition
ArmUser		ArmMetricGaugeFloat32
		ArmMetricGaugeFloat32Definition
		ArmMetricGroup
		ArmMetricGroupDefinition
		ArmMetricNumericId32
		ArmMetricNumericId32Definition
		ArmMetricNumericId64
		ArmMetricNumericId64Definition
		ArmMetricString32
		ArmMetricString32Definition
		ArmTranReportWithMetrics
		ArmTransactionWithMetrics
		ArmTransactionWithMetricsDefinition

9.2 Interface List (in Alphabetical Order)

The following table lists in alphabetical order all the interfaces that comprise the ARM specification, and lists the Java package in which they can be found. It also lists whether instances of the interface can be instantiated. Some interfaces cannot be instantiated because they define only constants or they are abstract superclasses. To create instances of instantiable interfaces, use the appropriate method of the factory class for the package.

Class Name (in Alphabetic Order)	Java Package	Can be Instantiated?
ArmApplication	org.opengroup.arm40.transaction	Yes
ArmApplicationDefinition	org.opengroup.arm40.transaction	Yes
ArmApplicationRemote	org.opengroup.arm40.tranreport	Yes
ArmConstants	org.opengroup.arm40.transaction	No
ArmCorrelator	org.opengroup.arm40.transaction	Yes
ArmErrorCallback	org.opengroup.arm40.transaction	Yes
ArmID	org.opengroup.arm40.transaction	Yes
ArmIdentityProperties	org.opengroup.arm40.transaction	Yes
ArmIdentityPropertiesTransaction	org.opengroup.arm40.transaction	Yes
ArmInterface	org.opengroup.arm40.transaction	No
ArmMetric	org.opengroup.arm40.metric	No
ArmMetricCounter32	org.opengroup.arm40.metric	Yes
ArmMetricCounter32Definition	org.opengroup.arm40.metric	Yes
ArmMetricCounter64	org.opengroup.arm40.metric	Yes
ArmMetricCounter64Definition	org.opengroup.arm40.metric	Yes
ArmMetricCounterFloat32	org.opengroup.arm40.metric	Yes
ArmMetricCounterFloat32Definition	org.opengroup.arm40.metric	Yes
ArmMetricDefinition	org.opengroup.arm40.metric	No
ArmMetricFactory	org.opengroup.arm40.metric	Yes
ArmMetricGauge32	org.opengroup.arm40.metric	Yes
ArmMetricGauge32Definition	org.opengroup.arm40.metric	Yes
ArmMetricGauge64	org.opengroup.arm40.metric	Yes
ArmMetricGauge64Definition	org.opengroup.arm40.metric	Yes
ArmMetricGaugeFloat32	org.opengroup.arm40.metric	Yes
ArmMetricGaugeFloat32Definition	org.opengroup.arm40.metric	Yes
ArmMetricGroup	org.opengroup.arm40.metric	Yes
ArmMetricGroupDefinition	org.opengroup.arm40.metric	Yes
ArmMetricNumericId32	org.opengroup.arm40.metric	Yes

Class Name (in Alphabetic Order)	**Java Package**	**Can be Instantiated?**
ArmMetricNumericId32Definition	org.opengroup.arm40.metric	Yes
ArmMetricNumericId64	org.opengroup.arm40.metric	Yes
ArmMetricNumericId64Definition	org.opengroup.arm40.metric	Yes
ArmMetricString32	org.opengroup.arm40.metric	Yes
ArmMetricString32Definition	org.opengroup.arm40.metric	Yes
ArmSystemAddress	org.opengroup.arm40.tranreport	Yes
ArmToken	org.opengroup.arm40.transaction	No
ArmTranReport	org.opengroup.arm40.tranreport	Yes
ArmTranReportFactory	org.opengroup.arm40.tranreport	Yes
ArmTranReportWithMetrics	org.opengroup.arm40.tranreport	Yes
ArmTransaction	org.opengroup.arm40.transaction	Yes
ArmTransactionDefinition	org.opengroup.arm40.transaction	Yes
ArmTransactionFactory	org.opengroup.arm40.transaction	Yes
ArmTransactionWithMetrics	org.opengroup.arm40.metric	Yes
ArmTransactionWithMetricsDefinition	org.opengroup.arm40.metric	Yes
ArmUser	org.opengroup.arm40.transaction	Yes

9.3 Method Naming Conventions

Every attempt has been made to adhere to the naming conventions commonly used in Java programs. Here is a list of a few to be aware of:

- **get()** or **getSomething()** returns a primitive value or a reference to an object or array. If it is a reference, the object (or array) to which it refers is treated as immutable. Treating the object or array as immutable means the ARM implementation will not change the data in the object or array. Some object types are entirely immutable, including String, ArmToken, and its subclasses (ArmCorrelator, ArmSystemAddress, ArmID), and ArmSystem. Some object types will not be changed by the ARM implementation, but could be changed by the application (an array or an object that implements ArmMetric).
- **set()** or **setSomething()** sets a primitive value or a reference to an object or array.
- **copySomething(**destination**)** copies the actual data to a byte array. It does not merely copy the reference.
- **isSomething()** returns a boolean.
- **newSomething()** creates a Something object and returns a reference to it.

9.4 org.opengroup.arm40.transaction.ArmApplication

ArmApplication represents an instance of an executing application. It provides an anchor point for associating ArmTransaction objects with the application instance. Instances of ArmApplication are created using the **newArmApplication()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Application definition. The metadata common to all instances of this application. The value must not be null.
- Group (optional). A group is a set of application instances that are treated as a group for some aspects of management, such as workload balancing (work could be routed to any instance of the group). The maximum length is 255 characters. The value may be null.
- Instance (optional). The instance could be used to distinguish between instances of the same application. ARM does not require that this field be used or that it be unique, though its use is suggested. The maximum length is 255 characters. The value may be null.
- Context values (optional). The “value” part of “name=value” properties that may vary per application instance. The “name” part is available via ArmApplicationDefinition→ArmIdentityProperties. The values are position-sensitive – they match the context name array in the referenced ArmIdentityProperties. If either the name or value of a property is null, both the name and value are considered to have a null value.

  Note that both ArmApplication and ArmTransaction have context values that may be unique for each application and transaction instance, respectively. However, the mechanisms for setting the context values are different. In ArmApplication, they are set in the factory method and are immutable afterwards. In ArmTransaction, they are set at any time and processed each time a **start()** executes.

  The reason for the difference is that ArmApplication objects are not reused. One object is created per application instance. If another instance starts, a new ArmApplication is created. This is a reasonable design pattern because applications are long-lived. In most cases, once an application starts, it continues to execute for a long time, so the cost of creating the ArmApplication object and then garbage collecting it is not significant.

  Transactions, on the other hand, are often very short-lived, with response times that are often measured in milliseconds. Creating a new ArmTransaction object for each transaction instance, and then garbage collecting it afterwards, would result in unacceptable and unnecessary overhead. Instead, an ArmTransaction object is created and then reused over and over. Each executing transaction is represented by the paired **start()**/**stop()** methods. For this reason, context values are set using setter methods.

end() indicates that the application instance has halted. After **end()** executes, the application instance may not call any other method of the ArmApplication object, it may not use any reference to the ArmApplication object, nor may it call any method of any object created using a reference to the ArmApplication object (e.g., creating an instance of ArmTransaction using the **newArmTransaction()** method of ArmTransactionFactory). Any transactions that are currently in-process [**start()** executed but **stop()** not executed] will be discarded by implicitly executing the ArmTransaction **reset()** method. The ARM implementation should protect itself against a poorly behaved application that does not respect the specification.

```
public interface ArmApplication extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int end();
    public String getContextValue(int index);
    public ArmApplicationDefinition getDefinition();
    public String getGroup();
    public String getInstance();
}
```

9.5 org.opengroup.arm40.transaction.ArmApplicationDefinition

ArmApplicationDefinition describes the attributes of an application that do not change from one instance of the application to another. It provides an anchor point for associating ArmTransactionDefinition and ArmMetricDefinition objects with the application. It is created with the **newArmApplicationDefinition()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Name. The maximum length is 127 characters (CIM allows 256 but ARM 4.0 C Bindings allow 128 characters, including the null-termination character, so 127 is used). The name must not be null or zero length. A name should be chosen that is unique, so generic names that might be used by a different development team, such as "Payroll Application", should not be used.

- (optional) Identity property names and values and context property names. They may be null.

- (optional) ID. An optional 16-byte ID may be associated with the identity of an application definition. The returned value, which could be null, is the same value passed to the **newArmApplicationDefinition()** method of ArmTransactionFactory. The ID value is bound to a unique combination of the application name, any identity property names and values, and any context property names. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

destroy() does not, of course, destroy the ArmApplicationDefinition object. It does signal to the ARM implementation that the definition and all related definitions (e.g., ArmTransactionDefinition) within its scope are no longer needed. The normal behavior would be for the ARM implementation to release its references to all those objects. If the application also releases its references, the objects would be eligible for garbage collection. After **destroy()** is called, no method on any object that is scoped by the ArmApplicationDefinition should be called again. If a method is called, the results are unpredictable.

```
public interface ArmApplicationDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public void destroy();
    public String getName();
    public ArmIdentityProperties getIdentityProperties();
    public ArmID getID();
}
```

9.6 org.opengroup.arm40.tranreport.ArmApplicationRemote

ArmApplicationRemote represents an instance of an application executing on a remote system. It differs from ArmApplication in that ArmApplication represents an application executing on the local system.

ArmApplicationRemote provides an anchor point for associating ArmTranReport objects with a system's network address. Instances of ArmApplicationRemote are created using the **newArmApplicationRemote()** method of ArmTranReportFactory. It adds the following attribute to those in ArmApplication:

- System address. The system address is the address (format and address byte array) of the system.

```
public interface ArmApplicationRemote extends ArmApplication {
// No Public Constructors
// Public Instance Methods
    public ArmSystemAddress getSystemAddress();
}
```

9.7 org.opengroup.arm40.transaction.ArmConstants

ArmConstants are constants that are widely used in the ARM 4.0 interfaces. They are declared in one interface as a matter of convenience (instead of duplicating them in multiple interfaces).

```
public interface ArmConstants {
// No Public Constructors
// Constants
    public static final int CORR_MAX_LENGTH;
        // Max length of a correlator (currently = 512 bytes)
    public static final int CORR_MIN_LENGTH;
        // Minimum length of a correlator (= 4 bytes)
    public static final int DIAG_DETAIL_MAX_LENGTH;
        // Maximum length of diagnostic detail string
        // (currently = 4095 characters)
    public static final int ID_LENGTH;
        // Length of all IDs (16 bytes)
    public static final int METRIC_MAX_COUNT;
        // Max number of metric slots (currently = 7)
    public static final int METRIC_MAX_INDEX;
        // Max index of a metric slot (currently = 6)
    public static final int METRIC_MIN_INDEX;
        // Min index of a metric slot (currently = 0)
    public static final int NAME_MAX_LENGTH;
        // Max chars in app/tran/metric name (currently = 127)
    public static final int PROPERTY_MAX_COUNT;
        // Maximum number of  identity and context properties
        // (currently = 20)
    public static final int PROPERTY_MAX_INDEX;
        // Maximum array index of an identity or context property
        // (currently = 19)
    public static final int PROPERTY_MIN_INDEX;
        // Minimum array index of an identity or context property
        // (currently = 0)
    public static final int PROPERTY_NAME_MAX_LENGTH;
        // Max chars in an identity or context property
        // (currently = 127)
    public static final int PROPERTY_URI_MAX_LENGTH;
        // Maximum chars in an URI property (currently = 4095)
    public static final int PROPERTY_VALUE_MAX_LENGTH;
        // Max chars in an identity or context property
        // (currently = 255)
    public static final int STATUS_ABORT;
        // Valid status value for ArmTranReport and ArmTransaction (=1)
    public static final int STATUS_FAILED;
        // Valid status value for ArmTranReport and ArmTransaction (=2)
    public static final int STATUS_GOOD;
        // Valid status value for ArmTranReport and ArmTransaction (=0)
    public static final int STATUS_INVALID;
        // Status value used when appl. passes an invalid value (=-1)
    public static final int STATUS_UNKNOWN;
        // Valid status value for ArmTranReport and ArmTransaction (=3)
    public static final int USE_CURRENT_TIME;
        // Used with ArmTranReport (= -1)
// No Instance Methods
}
```

9.8 org.opengroup.arm40.transaction.ArmCorrelator

ArmCorrelator represents a correlation token passed from a calling transaction to a called transaction. The correlation token may be used to establish a calling hierarchy across processes and systems. A correlator contains a two-byte length field, a one-byte format ID, a one-byte flag field, plus it may contain other data that is used to uniquely identify an instance of a transaction. Applications do not need to understand correlator internals. See Appendix B for more information about correlator formats.

A correlation token is a maximum of 512 bytes, including the header.

ArmCorrelator is created in one of three ways:

- The **newArmCorrelator()** method of ArmTransactionFactory takes as input a byte array in network byte order, such as would be received from a caller.
- The **getCorrelator()** method of ArmTransaction creates a correlator object for the currently or most recently executed transaction.
- The **generateCorrelator()** method of ArmTranReport creates a correlator object for what is presumed to be the next transaction to be executed.

An application may extract the byte array in network byte order, which is the format needed to send to a called transaction, using the **copyBytes()** or **getBytes()** methods of ArmToken, ArmCorrelator's parent interface.

Interface methods:

- **isAgentTrace()** indicates whether the "agent trace" flag is on in the correlator.
- **isApplicationTrace()** indicates whether the "application trace" flag is on in the correlator.

```
public interface ArmCorrelator extends ArmToken {
// No Public Constructors
// Public Instance Methods (in addition to those defined in ArmToken)
// (Implementations should also override equals() and hashCode()
// from java.lang.Object.)
    public boolean isAgentTrace();
    public boolean isApplicationTrace();
}
```

9.9 org.opengroup.arm40.transaction.ArmErrorCallback

The use of ArmErrorCallback is *optional*.

ArmErrorCallback is different from all the other ARM interfaces because instead of the ARM implementation creating objects that implement the interface, the *application* creates an object that implements it. The application can create an ArmErrorCallback and register it with the ArmApplication **setErrorCallback()** method. If the registration is accepted, anytime a method results in an error, the ARM implementation:

- Sets the object's error code. This error code can be retrieved with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive.
- Calls **errorCodeSet()** in ArmErrorCallback, if an ArmErrorCallback has been successfully registered.
- Returns the error code if the method that caused the error returns an error code.

There are no expected or required behaviors for the implementation of **errorCodeSet()**. Examples of things a callback method may do are:

- Nothing.
- Log an error. This would be a common behavior during program debug.
- Reset the error code to zero, or set it to some other value. This is a way for an error handling policy to be implemented centrally, and communicated to the rest of the application code.

When **errorCodeSet()** is called, all the parameters must contain non-null values.

- errorObject is a reference to the ARM implementation's object that detected the error. The callback method can use the errorObject **getErrorCode()** to get the error code value, which will be negative.
- interfaceName is the name of an interface in one of the ARM specification packages.
- methodName is the name of a method in that interface.

```
public interface ArmErrorCallback {
// No Public Constructors
// Public Instance Methods
    public void errorCodeSet(ArmInterface errorObject,
        String interfaceName, String methodName);
}
```

9.10 org.opengroup.arm40.transaction.ArmID

ArmID implements an immutable wrapper around a 16-byte ID. IDs may be used to identify metadata about applications, transactions, metrics, systems, and users. The ID may be a standard DCE UUID (universally unique identifier) but need not be. Any unique 16-byte value will suffice. There is no central registry of IDs that would guarantee uniqueness. Programs creating these IDs are expected to use an algorithm that will take advantage of the available 128 bits to create an ID for which there will be a vanishingly small probability of its being a duplicate of an ID created by another program.

It is created with the **newArmID()** method of ArmTransactionFactory, whose input is the 16 bytes in a byte array.

This interface was named ArmUUID in ARM 3.0. There are no other changes for ARM 4.0.

```
public interface ArmID extends ArmToken {
// No Public Constructors
// No Public Instance Methods (see ArmToken for applicable methods)
// (Implementations should override equals() and hashCode()
// from java.lang.Object.)
}
```

9.11 org.opengroup.arm40.transaction.ArmIdentityProperties

ArmIdentityProperties is new in ARM 4.0. It addresses a requirement to accept a set of string name=value pairs that extend the concept of application and transaction identity and context.

ARM defines two types of properties – identity and context. The difference between them is as follows:

- An identity property's name and value are the same for all instances of an application or transaction.
- A context property's name is the same for all instances of an application or transaction, but a context property's value may vary for each instance.

ArmIdentityProperties contains some attributes that are common to the identity of both an application and a transaction. The full identity is captured in ArmApplicationDefinition and ArmTransactionDefinition, which each adds other attributes to those listed here. All these attributes are immutable.

- A set of identity names (maximum of 20; maximum length of 127 characters)
- A set of identity values (maximum of 20; maximum length of 255 characters)

- A set of context names (maximum of 20; maximum length of 127 characters). The context values may change with each instance so they are not part of the identity.

ArmIdentityProperties is created with the **newArmIdentityProperties()** method of ArmTransactionFactory.

Interface methods:

- **getIdentityName()** returns the string (the name part of the name=value identity property) at the specified array index. The returned value will be null if either the name or value at the index is set to null.
- **getIdentityValue()** returns the string (the value part of the name=value identity property) at the specified array index. The returned value will be null if either the name or value at the index is set to null.
- **getContextName()** returns the string (the name part of the name=value context property) at the specified array index. The returned value may be null.

```
public interface ArmIdentityProperties extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public String getIdentityName(int index);
    public String getIdentityValue(int index);
    public String getContextName(int index);
}
```

9.12 org.opengroup.arm40.transaction.ArmIdentityPropertiesTransaction

ArmIdentityPropertiesTransaction extends ArmIdentityProperties for transactions by adding a URI property. Unlike the other identity properties, the URI property is implicitly named.

Like other identity properties, it is the same for all instances of the same transaction, and is immutable. In practice, this means that the URI value may be truncated from the URI that is actually used. In particular, parameters that are appended to the end of a URI will often be truncated because they are often different each time the URI is invoked. In this case, the parameters might be provided as part of the URI context value of an ArmTransaction object.

ArmIdentityPropertiesTransaction is created using the **newArmIdentityPropertiesTransaction()** method of ArmTransactionFactory.

Interface methods:

- **getURIValue()** returns the string representing the URI value, if any. The returned value will be null if the value is set to null.

```
public interface ArmIdentityPropertiesTransaction extends
    ArmIdentityProperties {
// No Public Constructors
// Public Instance Methods
    public String getURIValue();
}
```

9.13 org.opengroup.arm40.transaction.ArmInterface

ArmInterface is the root of the inheritance hierarchy for almost all ARM interfaces. It provides a common way to handle errors. If a method invocation on any ARM object causes an error, the error code returned by the object's **getErrorCode()** will be negative. If no error occurs, the error code is zero. Several methods also return the error code as an int return value. If an error occurs in a factory method (e.g., a method in ArmTransactionFactory), the error code is set in both the factory object and the newly created object. The implementation of the object has sole discretion as to whether a method results in an error.

The error code may change any time a method of the object is executed. Executing a method overrides the previous error code value. The only methods that will never change the error code are **getErrorCode()** and **getErrorMessage(**). If multiple threads are processing the same object simultaneously, the results are unpredictable. An error code set as a result of an operation in method X could be replaced by an operation in method Y before thread X executes **getErrorCode()**.

The error code can also be set with **setErrorCode()**. **setErrorCode()** is a way for the application to reset or change the error code. It would most typically be used in a callback registered with the application's ArmApplication. In this way the application can implement a central error handling policy in the callback. A value set with **setErrorCode()** in a callback overwrites the previous value (i.e., the value that caused the callback to be made).

For any non-zero error code returned by an object, the application can request from the same object a string message describing the error using **getErrorMessage()**. If the object does not support the function or does not recognize the error code, it returns null.

```
public interface ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int getErrorCode();
    public void setErrorCode(int errorCode);
    public String getErrorMessage(int errorCode);
}
```

9.14 org.opengroup.arm40.metric.ArmMetric

ArmMetric is a superclass for all the metric interfaces. The common behavior of all metric subclasses is:

- **getDefinition()** returns the descriptive metadata (name, units, usage, ID) about the metric. The returned object will be the appropriate subclass of ArmMetricDefinition.
- Each subclass is also expected to implement **get()** and **set()** methods that take and/or return data of the appropriate type for the subclass. These methods are not defined in this interface because they have different signatures, depending on the type of metric.

Objects that implement a subclass of ArmMetric are used with ArmTransactionWithMetrics and ArmTranReportWithMetrics. They are bound via ArmMetricGroup when the ArmTransactionWithMetrics or ArmTranReportWithMetrics instance is created. Each ArmMetric instance can be bound to any number of transaction instances. Setting the value of the ArmMetric instance effectively sets the value for all the transaction instances to which it is bound. The value affects each ArmTransactionWithMetrics instance the next time a **start()**, **update()**, or **stop()** is executed on the instance. The value affects each ArmTranReportWithMetrics instance the next time a **report()** is executed on the instance.

```
public interface ArmMetric extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricDefinition getDefinition();
}
```

9.15 org.opengroup.arm40.metric.ArmMetricCounter32

ArmMetricCounter32 implements a 32-bit integer counter. It is the same as ARM 2.0 metric type=1 (ARM_Counter32).

```
public interface ArmMetricCounter32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int get();
    public int set(int value);
}
```

9.16 org.opengroup.arm40.metric.ArmMetricCounter32Definition

ArmMetricCounter32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounter32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.17 org.opengroup.arm40.metric.ArmMetricCounter64

ArmMetricCounter64 implements a 64-bit integer counter. It is the same as ARM 2.0 metric type=2 (ARM_Counter64).

```
public interface ArmMetricCounter64 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long get();
    public int set(long value);
}
```

9.18 org.opengroup.arm40.metric.ArmMetricCounter64Definition

ArmMetricCounter64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounter64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.19 org.opengroup.arm40.metric.ArmMetricCounterFloat32

ArmMetricCounterFloat32 implements a 32-bit floating-point counter. It is roughly equivalent to the ARM 2.0 metric type=3 (ARM_CntrDivr32). Instead of providing two integer values that can be divided to produce a floating-point value, which is what was done in the C bindings for ARM 2.0 and ARM 4.0, a floating-point value is provided directly. This was not done with the C bindings because ARM would have to support multiple floating-point formats, depending on the programming language and/or machine architecture, and the complexity was not deemed worthwhile.

```
public interface ArmMetricCounterFloat32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public float get();
    public int set(float value);
}
```

9.20 org.opengroup.arm40.metric.ArmMetricCounterFloat32Definition

ArmMetricCounterFloat32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounterFloat32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.21 org.opengroup.arm40.metric.ArmMetricDefinition

ArmMetricDefinition is a superclass for all the metric definition interfaces. All the methods are defined in ArmMetricDefinition. The subclasses serve as markers for the data types.

All the publicly accessible attributes are immutable and have a getter method for them. The attributes are:

- The definition of the application that contains the transaction. It must not be null.
- Name. The maximum length is 127 characters. The name must not be null or zero length.
- Units. An optional string describing the units of measurement, such as "files" or "jobs in queue". It may be null.
- Usage. Describes any additional information about the semantics of the metric. Most metrics will be classified as "general", indicating that there is no specific semantic declared. The two specific semantics are the size of the transaction, such as the number of files that were backed up, and a status that could contain data, such as an error code that describes why a transaction failed. See the ArmMetricDefinition interface for constants for the defined value. Any negative value is also permitted; the negative range is reserved for application-specific values.
- Optional ID. An optional 16-byte ID may be associated with the identity of a metric definition. The returned value, which could be null, is the same value passed to the **newArmMetricDefinition()** method of ArmMetricFactory. The ID value is bound to a unique combination of the metric format (e.g., Counter32), name, usage, and unit properties. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

```
public interface ArmMetricDefinition extends ArmInterface {
// No Public Constructors
// Constants
    public static final int METRIC_USE_GENERAL;
        // No usage semantics are declared.
    public static final int METRIC_USE_TRAN_SIZE;
        // Metric represents the "size" of the transaction
        // (counter and gauge only).
    public static final int METRIC_USE_TRAN_STATUS;
        // Metric represents status, like an error code
        // (numeric ID and string only).
// Public Instance Methods (subclasses should also declare get(),
// new(), and set() for the appropriate data types).
    public ArmApplicationDefinition getApplicationDefinition();
    public ArmID getID();
    public String getName();
    public String getUnits();
    public short getUsage();
}
```

9.22 org.opengroup.arm40.metric.ArmMetricFactory

ArmMetricFactory provides methods to create instances of the classes in the org.opengroup.arm40.metric package.

newArmMetricCounter32Definition
newArmMetricCounter64Definition
newArmMetricCounterFloat32Definition

newArmMetricGauge32Definition
newArmMetricGauge64Definition
newArmMetricGaugeFloat32Definition
newArmMetricNumericId32Definition
newArmMetricString32Definition

Creates the metadata for the respective data type. All metrics have the same metadata. See the ArmMetricDefinition interface description for details.

newArmMetricGroupDefinition()

Creates an ordered set of ArmMetricDefinition subclasses ready for binding to transaction definition objects. The input is an array of ArmMetricDefinition objects. The ordering in the array is important. The array can have up to seven elements and is position-sensitive. To remain consistent with ARM 2.0, any ArmMetricDefinition subclass except ArmMetricString32Definition can be assigned to elements 0:5 and only ArmMetricString32Definition can be assigned to element 6.

Any element can be null. If the input array has fewer than seven elements, the rest of the elements are assigned a value of null. The array can be sparsely populated. For example, there can be a non-null ArmMetricDefinition reference in element 0 and 6, and null references in elements 1:5.

newArmTransactionWithMetricsDefinition()

Creates an object that contains the metadata about a transaction. Details about the metadata are found in the ArmTransactionDefinition section. In addition, a binding to a set of metadata about metrics is added via an ArmMetricGroupDefinition object.

newArmMetricCounter32
newArmMetricCounter64
newArmMetricCounterFloat32
newArmMetricGauge32
newArmMetricGauge64
newArmMetricGaugeFloat32
newArmMetricNumericId32
newArmMetricNumericId64
newArmMetricString32

Creates a metric of the specified type, using the input metadata.

newArmMetricGroup()

Creates an ordered set of ArmMetric subclasses ready for binding to transaction objects. The input is an array of ArmMetric objects plus a metric group definition object. The ordering in the array is important. The array can have up to seven elements and is position-sensitive. To remain consistent with ARM 2.0, any ArmMetric subclass except ArmMetricString32 can be assigned to elements 0:5 and only ArmMetricString32 can be assigned to element 6. The ArmMetricDefinition objects associated with each ArmMetric object must have the exact same values as the ArmMetricDefinition objects associated with the ArmMetricGroupDefinition object (they will often be the same objects, though this is not mandatory).

Any element can be null. If the input array has fewer than seven elements, the rest of the elements are assigned a value of null. The array can be sparsely populated. For example, there can be a non-null ArmMetric reference in element 0 and 6, and null references in elements 1:5.

newArmTranReportWithMetrics()
Creates an object that represents an instance of a transaction. The metadata about the transaction is supplied in an ArmTransactionWithMetricsDefinition object. An application instance (ArmApplication) provides a scoping context. The metrics are bound via an ArmMetricGroup object. If the metric group reference is null, the resulting ArmTransactionWithMetrics has no metrics, so it would have no more functions than an ArmTransaction object, except to return null to the **getMetricGroup()** method.

newArmTransactionWithMetrics()
Creates an object that represents an instance of a transaction. The metadata about the transaction is supplied in an ArmTransactionWithMetricsDefinition object. An application instance (ArmApplication) provides a scoping context. The metrics are bound via an ArmMetricGroup object. If the metric group reference is null, the resulting ArmTransactionWithMetrics has no metrics, so it would have no more functions than an ArmTransaction object, except to return null to the **getMetricGroup()** method.

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all zeros. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct; that is, any of its methods can be invoked without causing an exception, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 7 for a more complete explanation.

ArmMetricFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(**null**)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(**null**)** is executed. For a broader discussion of error handling, refer to Chapter 7.

Note that in the interface description below, the methods are not strictly alphabetical like they are with all other interfaces. This is because there are some logical groupings of methods, and it was felt that it would be more intuitive to present them in the groups.

```
public interface ArmMetricFactory extends ArmInterface {
// Public Constants
```

```
    public static final String propertyKey;
// Public Instance Methods
    /* --- metric definitions --- */
    public ArmMetricCounter32Definition
        newArmMetricCounter32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricCounter64Definition
        newArmMetricCounter64Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricCounterFloat32Definition
        newArmMetricCounterFloat32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGauge32Definition newArmMetricGauge32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGauge64Definition newArmMetricGauge64Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGaugeFloat32Definition
        newArmMetricGaugeFloat32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricNumericId32Definition
        newArmMetricNumericId32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricNumericId64Definition
        newArmMetricNumericId64Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricString32Definition newArmMetricString32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);

    /* --- other definitions --- */
    public ArmMetricGroupDefinition newArmMetricGroupDefinition(
        ArmMetricDefinition[] definitions);
    public ArmTransactionWithMetricsDefinition
        newArmTransactionWithMetricsDefinition(
        ArmApplicationDefinition app,
        String name,
        ArmIdentityPropertiesTransaction identityProperties,
        ArmMetricGroupDefinition definition,
        ArmID id);

    /* --- metric instances --- */
    public ArmMetricCounter32 newArmMetricCounter32(
        ArmMetricCounter32Definition definition);
    public ArmMetricCounter64 newArmMetricCounter64(
        ArmMetricCounter64Definition definition);
    public ArmMetricCounterFloat32 newArmMetricCounterFloat32(
        ArmMetricCounterFloat32Definition definition);
    public ArmMetricGauge32 newArmMetricGauge32(
        ArmMetricGauge32Definition definition);
    public ArmMetricGauge64 newArmMetricGauge64(
        ArmMetricGauge64Definition definition);
    public ArmMetricGaugeFloat32 newArmMetricGaugeFloat32(
        ArmMetricGaugeFloat32Definition definition);
    public ArmMetricNumericId32 newArmMetricNumericId32(
        ArmMetricNumericId32Definition definition);
    public ArmMetricNumericId64 newArmMetricNumericId64(
        ArmMetricNumericId64Definition definition);
    public ArmMetricString32 newArmMetricString32(
        ArmMetricString32Definition definition);
    /* --- other interfaces --- */
    public ArmMetricGroup newArmMetricGroup(
        ArmMetricGroupDefinition groupDefinition,
        ArmMetric[] metrics);
    public ArmTranReportWithMetrics newArmTranReportWithMetrics(
        ArmApplication app,
        ArmTransactionWithMetricsDefinition definition,
        ArmMetricGroup group);
    public ArmTransactionWithMetrics newArmTransactionWithMetrics(
        ArmApplication app,
        ArmTransactionWithMetricsDefinition definition,
        ArmMetricGroup group);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

9.23 org.opengroup.arm40.metric.ArmMetricGauge32

ArmMetricGauge32 implements a 32-bit integer gauge. It is the same as ARM 2.0 metric type=4 (ARM_Gauge32).

```
public interface ArmMetricGauge32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int get();
    public int set(int value);
}
```

9.24 org.opengroup.arm40.metric.ArmMetricGauge32Definition

ArmMetricGauge32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGauge32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.25 org.opengroup.arm40.metric.ArmMetricGauge64

ArmMetricGauge64 implements a 64-bit integer gauge. It is the same as ARM 2.0 metric type=5 (ARM_Gauge64).

```
public interface ArmMetricGauge64 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long get();
    public int set(long value);
}
```

9.26 org.opengroup.arm40.metric.ArmMetricGauge64Definition

ArmMetricGauge64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGauge64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.27 org.opengroup.arm40.metric.ArmMetricGaugeFloat32

ArmMetricGaugeFloat32 implements a 32-bit floating-point gauge. It is roughly equivalent to the ARM 2.0 metric type=6 (ARM_GaugeDivr32). Instead of providing two integer values that can be divided to produce a floating-point value, which is what was done in the C bindings for ARM 2.0 and ARM 4.0, a floating-point value is provided directly. This was not done with the C bindings because ARM would have to support multiple floating-point formats, depending on the programming language and/or machine architecture, and the complexity was not deemed worthwhile.

```
public interface ArmMetricGaugeFloat32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public float get();
    public int set(float value);
}
```

9.28 org.opengroup.arm40.metric.ArmMetricGaugeFloat32Definition

ArmMetricGaugeFloat32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGaugeFloat32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.29 org.opengroup.arm40.metric.ArmMetricGroup

ArmMetricGroup is used to bind objects that implement a subclass of ArmMetric to an ArmTransactionWithMetrics or ArmTranReportWithMetrics object. The binding occurs when the transaction object is created [using **newArmTransactionWithMetrics()** or **newArmTransReportWithMetrics()**] and is immutable afterwards.

index is the index into the ArmMetric array. It must have a value in the range 0:6. To remain consistent with ARM 2.0, any ArmMetric subclass except ArmMetricString32 can be assigned to elements 0:5 and only ArmMetricString32 can be assigned to element 6.

getDefinition()
Returns the metric group definition used to create this object.

getMetric(index**)**
Returns the metric at the array index. This value may be null.

isMetricValid()
Indicates whether an ArmMetric subclass at this array index is valid.

setMetricValid()

Indicates whether an ArmMetric subclass at this array index is valid when any of the following calls are made. If the valid flag is set then the metric value is processed.

ArmTranReportWithMetrics: **report()**

ArmTransactionWithMetrics: **start()**, **update()**, **stop()**

```
public interface ArmMetricGroup extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricGroupDefinition getDefinition();
    public ArmMetric getMetric(int index);
    public boolean isMetricValid(int index);
    public int setMetricValid(int index, boolean value);
}
```

9.30 org.opengroup.arm40.metric.ArmMetricGroupDefinition

ArmMetricGroupDefinition is used to bind ArmMetricDefinition objects to an ArmTransactionWithMetricsDefinition or ArmTranReportWithMetricsDefinition object. The binding occurs when the transaction object is created using **newArmTransactionWithMetricsDefinition()** or **newArmTranReportWithMetricsDefinition()** and is immutable afterwards.

index is the index into the ArmMetricDefinition array. It must have a value in the range 0:6. To remain consistent with ARM 2.0, any ArmMetricDefinition subclass except ArmMetricString32Definition can be assigned to elements 0:5 and only ArmMetricString32Definition can be assigned to element 6.

get(index**)** returns the metric at the array index. This value may be null.

```
public interface ArmMetricGroupDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricDefinition getMetricDefinition(int index);
}
```

9.31 org.opengroup.arm40.metric.ArmMetricNumericId32

ArmMetricNumericId32 implements a 32-bit integer numeric ID. It is the same as ARM 2.0 metric type=7 (ARM_NumericID32).

```
public interface ArmMetricNumericId32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int get();
    public int set(int value);
}
```

9.32 org.opengroup.arm40.metric.ArmMetricNumericId32Definition

ArmMetricNumericId32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricNumericId32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.33 org.opengroup.arm40.metric.ArmMetricNumericId64

ArmMetricNumericId64 implements a 64-bit integer numeric ID. It is the same as ARM 2.0 metric type=8 (ARM_NumericID64).

```
public interface ArmMetricNumericId64 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long get();
    public int set(long value);
}
```

9.34 org.opengroup.arm40.metric.ArmMetricNumericId64Definition

ArmMetricNumericId64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricNumericId64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.35 org.opengroup.arm40.metric.ArmMetricString32

ArmMetricString32 implements a string of 1 to 32 characters. It is similar to the ARM 2.0 metric type=10 (ARM_String32), with two differences:

- The characters are in the Java standard UCS-2 format.
- The limit of 32 in the ARM 2.0 C language interface is a *byte* limit. The limit in the ARM 4.0 C bindings is a *character* limit. Because a character may be represented by more than

one byte (e.g., a character in UTF-8 is represented as 1, 2, or 3 bytes), the ARM 4.0 C metric may be longer than 32 bytes.

```
public interface ArmMetricString32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public String get();
    public int set(String s);
}
```

9.36 org.opengroup.arm40.metric.ArmMetricString32Definition

ArmMetricString32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricString32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

9.37 org.opengroup.arm40.tranreport.ArmSystemAddress

ArmSystemAddress encapsulates the network addressing information for a system. It may be used with ArmTranReport if the reported transaction executed on a different system.

- Address. The system address is the network name or address of the system, as it would be sent in a data frame across a network in network byte order.
- Format. The format of the system address, such as an SNA address or a hostname.
- ID. An optional 16-byte ID associated with the format and address, if any. The ID value is bound to a unique combination of the format and address. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

equals(Object obj**)**, a method inherited from java.lang.Object, returns true if the internal data is byte-for-byte identical in two objects. For example, **a.equals(**b**)** returns true if and only if:

- Both a and b implement ArmSystemAddress.
- The inherited methods **a.getBytes()** and **b.getBytes()** would return byte arrays of identical lengths and contents.
- **a.getFormat()** and **b.getFormat()** would return identical values.

getAddress() returns a byte array containing the address. The returned value is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

getFormat() returns a short containing the format. The returned value is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

getID() returns the optional 16-byte ID associated with the format and address, if any. The returned value, which could be null, is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

The fields are set using the **newArmSystemAddress()** method of ArmTranReportFactory or the **getArmSystemAddress()** method of ArmSystem. There are no setter methods for the individual fields. The object is immutable.

```
public interface ArmSystemAddress extends ArmToken {
// public constants
    public static final short FORMAT_HOSTNAME;
    public static final short FORMAT_IPV4;
    public static final short FORMAT_IPV4PORT;
    public static final short FORMAT_IPV6;
    public static final short FORMAT_IPV6PORT;
    public static final short FORMAT_SNA;
    public static final short FORMAT_X25;
// No Public Constructors
// Public Instance Methods (in addition to those defined by ArmToken).
// (Implementations should also override equals() and hashCode() from
// java.lang.Object.)
    public byte[] getAddress();
    public short getFormat();
    public ArmID getID();
}
```

9.38 org.opengroup.arm40.transaction.ArmToken

ArmToken is an abstract interface which is a superclass of ArmCorrelator, ArmSystemAddress (in the tranreport package), and ArmID, expressing the common part of their interfaces. ArmToken is abstract in the sense that any ArmToken object returned by any method in this specification satisfies one of the subclass interfaces. Objects of these identify particular entities. These objects contain a byte array data token, plus optionally other identifying data, which together comprise the value of the object. The data token is always immutable. Subclasses of ArmToken can be thought of as wrappers around the data token.

The methods are as follows:

- **copyBytes()** is used to copy the token to a byte array that is already allocated. There are two forms. Both return a boolean. If the return value is true, the operation was successful. If the return value is false, the operation was not successful and the contents of the target array are undetermined. The most likely errors are an attempt to copy into a null pointer or into an array that is not long enough to hold the entire token.

  — **copyBytes(**byte[] dest**)** copies the token's byte array into the destination byte array, which must have a length greater than or equal to the token's byte array.

— **copyBytes(**byte[] dest, int offset**)** copies the token's byte array into the destination byte array at the specified offset. The destination array must be large enough to hold the byte array value; that is, dest.length-offset >= **token.getLength()**.

- **getBytes()** returns a newly allocated byte array into which the token is copied. It is equivalent to creating a byte array of length **getLength()** and then executing **copyBytes()** into the new array. The ARM implementation would typically not keep a reference to the array, because that would interfere with garbage collection.
- **getLength()** returns the size of the byte array part of the token.

To make it possible to compare the values of these tokens, and to use these tokens as hashkeys (so that the user can associate data with a particular token), this specification requires that any class implementing any of these types override the java.lang.Object methods **equals(**Object**)** and **hashCode()**. The behavior of these methods must be the following.

a.equals(b**)** returns true if the ArmToken objects a and b have the same value (internal data is byte-for-byte identical in two objects). That is, **a.equals(**b**)** is true if and only if:

- a and b implement the same interface ArmCorrelator, ArmSystemAddress, or ArmID.
- **a.getBytes()** and **b.getBytes()** would return byte arrays of identical lengths and contents.
- If a subclass of ArmToken defines other data values (specifically, ArmSystemAddress defines a short field named format), these other data values are all identical.

If **a.equals(**b)==true, **a.hashCode()** and **b.hashCode()** will return the same value. The **hashCode()** value is implementation-defined. In other words, hashcode values are not necessarily portable. A hashcode generated on one system by one implementation may not equal a hashcode value generated on another system by a different implementation, even if **a.equals(**b)==true.

```
public interface ArmToken extends ArmInterface {
// No Public Constructors
// Public Instance Methods
// (Subclasses should also override equals() and hashCode() from
// java.lang.Object.)
    public boolean copyBytes(byte[] dest);
    public boolean copyBytes(byte[] dest, int offset);
    public byte[] getBytes();
    public int getLength();
}
```

9.39 org.opengroup.arm40.tranreport.ArmTranReport

ArmTranReport is similar to ArmTransaction. Both are used to provide data about executing transactions. Instances of both are created based on metadata represented by an ArmTransactionDefinition, which in turn is scoped by an application definition. Both are scoped by a running application instance, represented by ArmApplication. There are two fundamental differences:

- With ArmTransaction, the response time is measured based on **start()** and **stop()** events. With ArmTranReport, the application measures the response time, and reports it with a single **report()** event.
- With ArmTransaction, the transaction always executes on the local system in the same JVM (Java Virtual Machine). With ArmTranReport, the transaction may execute in the same JVM, in a different JVM on the same system, or on a different system.

When executing in the same JVM, the ArmTranReport object is created with an ArmApplication. When executing in a different JVM on the same system or on a different system, the ArmTranReport object is created with an ArmApplicationRemote (a subclass of ArmApplication).

The two key methods of ArmTranReport are **generateCorrelator()** and **report()**.

- **generateCorrelator()** generates a new correlator using the immutable data set in the factory method and the current property values set by the four setter methods. It is assumed that **generateCorrelator()** is executed zero or once per transaction instance. The practical ramification is that the method implementation will update its internal state to have a unique identifier for an instance (the equivalent of ArmTransaction's start handle). **getCorrelator()** returns the most recently generated ArmCorrelator or null, if **generateCorrelator()** has never been executed.
- **report()** is used to provide measurements about a completed transaction. There are two forms. Both provide the status (one of the STATUS_* constants in ArmConstants) and the response time. One also provides a stop time in the form of milliseconds since January 1, 1970, which is the same format returned by **java.lang.System.currentTimeMillis()**. If a stop time is not provided, or a stop time of –1 (USE_CURRENT_TIME) is provided, the ARM implementation substitutes the current time; that is, the time when the **report()** method executes. The optional form that takes a string is a way for an application to provide additional diagnostic details when the status is something other than STATUS_GOOD.

As noted above, **generateCorrelator()** updates the internal state for a new transaction instance. The first time **report()** executes after **generateCorrelator()**, **report()** will not update the internal state for a new transaction instance; it will use the instance identifier from the **generateCorrelator()**. If **report()** executes twice in succession, or if **generateCorrelator()** has never been executed, **report()** will update the internal state for a new transaction instance. Summarizing, there are two patterns:

- If correlators for the current transaction are not requested, **generateCorrelator()** is not used. **report()** is executed after each instance completes, and each time it generates a new instance identifier, like a start handle.
- If correlators for the current transaction are requested, **generateCorrelator()** and **report()** are used in pairs. First **generateCorrelator()** establishes the transaction instance identifiers, as well as creating a correlator. This correlator is sent to downstream transactions. After the downstream transactions complete, and the current instance completes, **report()** provides the measurements. In this case **report()** does not update the transaction instance identifier.

In addition to the identity properties from ArmApplication and ArmTransactionDefinition, there are four optional setter methods to establish additional instance-level context. They can be used at any time to update the attribute within the object. The only time the properties are meaningful is when **generateCorrelator()** or **report()** executes. At the moment either method executes, the current values are used, any or all of which may be null.

- **setContextURIValue()** sets the URI context value. **getContextURIValue()** returns the value.
- **setContextValue()** sets one of the maximum 20 context properties that may change for each transaction instance. **getContextValue()** returns the value, whether null or not. The context property name at the specified array index must have been set to a non-null value when the ArmTransactionDefinition object was created. If the name is null, the value will be set to null.
- **setParentCorrelator()** sets the correlator of the parent transaction. **getParentCorrelator()** returns the value, whether null or not.
- **setUser()** associates a user, represented by an instance of ArmUser, to the ArmTranReport instance. This user is assumed to be the user for all **start()**/**stop()** pairs until the association is changed or cleared. **setUser(**null**)** clears any existing association to an ArmUser. **getArmUser()** returns the current value, whether null or not.

There are two methods that return the data used in the factory method to create this instance of ArmTranReport: **getApplication()** and **getDefinition()**.

```
public interface ArmTranReport extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmCorrelator generateCorrelator();
    public ArmApplication getApplication();
    public String getContextURIValue();
    public String getContextValue(int index);
    public ArmCorrelator getCorrelator();
    public ArmCorrelator getParentCorrelator();
    public long getResponseTime();
    public int getStatus();
    public ArmTransactionDefinition getDefinition();
    public ArmUser getUser();
    public int report(int status, long respTime);
    public int report(int status, long respTime, long stopTime);
    public int report(int status, long respTime, String
        diagnosticDetail);
    public int report(int status, long respTime, long stopTime, String
        diagnosticDetail);
    public int setContextURIValue(String value);
    public int setContextValue(int index, String value);
    public int setParentCorrelator(ArmCorrelator parent);
    public int setUser(ArmUser user);
}
```

9.40 org.opengroup.arm40.tranreport.ArmTranReportFactory

ArmTranReportFactory provides methods to create instances of the classes in the org.opengroup.arm40.tranreport package.

newArmApplicationRemote()
Creates an ArmApplicationRemote. See the ArmApplicationRemote description for details about the parameters. If systemAddress is null, the addressing information for the local system is used.

newArmSystemAddress()
Creates an ArmSystemAddress from the specified format and the input byte array. See the ArmSystemAddress description for details about the parameters.

newArmTranReport()
Creates an object that represents an instance of a transaction. The metadata is supplied in an ArmTransactionDefinition object. It is scoped by an application instance, represented by ArmApplication (or its subclass, ArmApplicationRemote).

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all zeros. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct; that is, any of its methods can be invoked without causing an exception, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 7 for a more complete explanation.

ArmTranReportFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(null)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(null)** is executed. For a broader discussion of error handling, refer to Chapter 7.

```
public interface ArmTranReportFactory extends ArmInterface {
// Public Constants
    public static final String propertyKey;
// Public Instance Methods
    public ArmApplicationRemote newArmApplicationRemote(
        ArmApplicationDefinition definition,
        String group, String instance,
        String[] contextValues,
        ArmSystemAddress systemAddress);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        ArmID id);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        int offset,
        ArmID id);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        int offset,
        int length,
        ArmID id);
    public ArmTranReport newArmTranReport(
        ArmApplication app,
        ArmTransactionDefinition definition);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

9.41 org.opengroup.arm40.metric.ArmTranReportWithMetrics

ArmTranReportWithMetrics is a subclass of ArmTranReport that is used if the application wishes to use metrics. All the ArmTranReport semantics for using **report()** apply to ArmTranReportWithMetrics.

It extends ArmTranReport by adding methods to manipulate metrics. The ArmMetric subclass objects are bound to an ArmTranReportWithMetrics object when the ArmTranReportWithMetrics is created. This is done by specifying ArmMetricGroup in the **newArmTranReportWithMetrics()** method of ArmMetricFactory.

getTransactionWithMetricsDefinition()

Returns the ArmTransactionWithMetricsDefinition object that contains the metadata describing this transaction, including the metric definitions.

getMetricGroup()

Returns the ArmMetricGroup object that was bound when ArmTransactionWithMetrics is created. The returned value may be null.

```
public interface ArmTranReportWithMetrics extends ArmTranReport {
// No Public Constructors
// Public Instance Methods
    public ArmTransactionWithMetricsDefinition
        getTransactionWithMetricsDefinition();
    public ArmMetricGroup getMetricGroup();
}
```

9.42 org.opengroup.arm40.transaction.ArmTransaction

For most applications, ArmTransaction is the most important of all the ARM classes, and the most frequently used. Many applications operate only on ArmTransaction objects after some initialization (using ArmTransactionFactory, ArmApplicationDefinition, ArmApplication, and ArmTransactionDefinition). Instances of ArmTransaction represent transactions when they execute. A "transaction" is any unit of work that has a clearly understood beginning and ending point, and which begins and ends in the same operating system process. Examples include a remote procedure call, a database transaction, and a batch job. It is not necessary that an ARM transaction implement robust functions such as commit and rollback.

The application creates as many instances as it needs. This will typically be at least as many as the number of transactions that can be executing simultaneously. An application may create a pool of ArmTransaction objects, take one from the pool to use when a transaction starts, and put it back in the pool after the transaction ends for later reuse. Another strategy is to create one instance of each type per thread, which eliminates the need to manage the pool, handle synchronization if the pool is depleted, etc.

ArmTransaction is created with the **newArmTransaction()** method of ArmTransactionFactory. The metadata common to all instances is contained in the ArmTransactionDefinition used to create the object. Each transaction is scoped by an application instance, represented by ArmApplication.

The most frequently (and often the only) used methods are **start()**, **getCorrelator()**, and **stop()**. A typical sequence is as follows:

- Just prior to executing a transaction, such as a remote procedure call, call **start()** to signal to ARM that the measurable transaction is beginning. **start()** causes ARM to capture the current time. If a correlation token was received when your program was invoked, pass it as a parameter on the **start()**.
- Just after executing **start()**, and just prior to executing the transaction, call **getCorrelator()** to get a correlation token that can be sent along with the other transaction parameters to the receiver. Not all programs use correlators, but their use is highly recommended, because without them, it is usually impossible to drill down to understand the components of a transaction, where they executed, what resources they used, etc.
- Execute the transaction (e.g., make the remote procedure call).
- As soon as it ends, call **stop()**, passing a status to indicate whether the transaction succeeded. **stop()** causes ARM to capture the current time. The response time is determined by calculating the duration between the **start()** and **stop()** events.

Following are details about each method. All methods that return an int are returning an error code that the application may but need not test. Refer to Chapter 7 for more information about handling errors.

- **start()** indicates when a transaction begins. Because the response time depends on when **start()** executes, it should execute as close to the actual start time as possible. After **start()** executes, it should not be executed again until **reset()** or **stop()** is executed. If **start()** executes consecutively, the behavior is undefined.

There are four versions of **start()**, depending on whether a parent correlator is provided, and if one is provided, the format of the input data. The length of the correlator is in the first two bytes of the correlator byte array, with the bytes in network byte order. When the input is a byte array, the length of the array does not matter, as long as it is at least long enough to hold the correlator, based on the two-byte length field.

- After a **start()** there can be any number of **update()** calls until a **stop()**. If it is executed at any other time, it is ignored. The behavior of **update()** issued at any other time is undefined. **update()** is used for two purposes:

  — It serves as a heartbeat to show that a transaction is still executing. This is especially useful if the transaction is a long-running job.

  — When used with ArmTransactionWithMetrics, a subclass of ArmTransaction (in the org.opengroup.arm4.metrics package), any of the metric values can be provided with an **update()**.

- **stop()** indicates when a transaction ends and what the status of the transaction was. Because the response time depends on when **stop()** executes, it should execute as close to the actual stop time as possible. If **stop()** is erroneously issued when there is no transaction active [**start()** issued without a matching **stop()**], it is ignored. The status must be one of STATUS_ABORT, STATUS_FAILED, STATUS_GOOD, or STATUS_UNKNOWN (all defined in ArmConstants). The optional form that takes a string is a way for an application to provide additional diagnostic details when the status is something other than STATUS_GOOD.

- **reset()** can be executed at any time. If a transaction is currently executing [**start()** executed without a matching **stop()**], the current transaction is discarded and treated as if the **start()** never executed. If no transaction is currently executing, the state of the object is unchanged. If there is any doubt about the state of an object, **reset()** gets the object into a known state in which a **start()** may be executed. **reset()** clears the arrival time and the current correlator; it does not change traceRequested or any of the context URI, context values, or user.

- **getCorrelator()** returns a reference to the correlator for the current transaction. It may be a newly created object. It can be executed anytime after **start()** is executed. Each time it is executed, it will return the same value until the next **stop()** or **reset()** is executed. If it is executed at any other time, it will return an ArmCorrelator object, but the data within the ArmCorrelator object is undefined and should not be used.

- **setTraceRequested()** is used to suggest or withdraw a suggestion from an application that a transaction be traced. **isTraceRequested()** is used to query the current trace request state. The initial state is false. Once set, it remains in that state until set to a different state.

- **bindThread()** and **unbindThread()** can be called from any thread to indicate that the thread is executing on behalf of the transaction instance. This is useful when multiple threads execute the same logical (ARM) transaction, because instrumentation of resource consumption at the thread level can be more precise. The thread remains bound to this transaction until **unbindThread()** is executed in this thread or **stop()** or **reset()** is executed.

- **block()** is used to indicate that the transaction instance is blocked waiting on an external transaction (which may or may not be instrumented with ARM) or some other event to complete. It has been found useful to separate out this "blocked" time from the elapsed time between the **start()** and **stop()**. **unblock()** indicates when the blocking condition has ended. A transaction may be blocked by multiple conditions simultaneously. A "block handle" returned by **block()** is the input parameter to **unblock()** to indicate which blocking condition has ended.

- **setArrivalTime()** can be used in situations in which the context of a transaction is not known when the transaction begins to execute, and for which there is a non-trivial delay before the context is known. ARM requires that the full context of a transaction be known when **start()** is executed (because the correlator is generated at this time). In ARM 2.0 and 3.0 there is no way to capture any time spent processing the transaction before the context is known. ARM 4.0 introduces the concept of an "arrival time". The "arrival time" is when processing of the transaction commenced. By default it is the moment in time when **start()** executes. If the delay between the start of processing and the execution of **start()** is significant, the application can capture the arrival time by invoking **setArrivalTime()**. This establishes a timestamp that will be used at the next **start()**, after which the value will be reset within the ArmTransaction object. The **reset()** and **stop()** methods also clear the value.

- **setContextURIValue()** sets the URI context value. **getContextURIValue()** returns the value. In most scenarios, a URI would be used as a transaction identity property or a context property, but not both. The only allowed exception is when the base part of the URI is used as an identity property, and the full URI (e.g., with the parameters) is used as a context property. Any other use of URIs as both identity and context properties is invalid.

- **setContextValue()** sets one of the maximum 20 context properties that may change for each transaction instance. **getContextValue()** returns the value. The context property name at the specified array index must have been set to a non-null value when the ArmTransactionDefinition object was created. If the name is null, the value will be set to null.

- **setUser()** associates a user, represented by an instance of ArmUser, to the ArmTransaction instance. This user is assumed to be the user for all **start()**/**stop()** pairs until the association is changed or cleared. **setUser(**null**)** clears any existing association to an ArmUser. **getUser()** returns the last value that was set.

- **getParentCorrelator()** returns the last value set on a **start()** method. If no value was set on the **start()** method, or if **start()** has never executed, it returns null.

- **getStatus()** returns the last value set on a **stop()** method. If **stop()** has never executed, it returns null.

- **getApplication()** returns the value passed to the **newArmTransaction()** method of ArmTransactionFactory.

- **getDefinition()** returns the value passed to the **newArmTransaction()** method of ArmTransactionFactory.

Note that there are some changes from ARM 3.0:

- Descriptive metadata has been moved from ArmTransaction to ArmApplicationDefinition and ArmTransactionDefinition.
- Three methods that were in ARM 3.0 are removed. These methods were almost never used. They also placed restrictions on implementations that could be unacceptable because they required an ARM implementation to return a value that had not been provided by the application. They can be contrasted with other getter methods that return data previously set by the application using ARM: **getRespTime(), getStopTime(), and getTranHandle().**
- The return type for **stop()** is changed from long to int. The long returned in ARM 3.0 was the response time, and the requirement to return it is considered unacceptable. The int return value is an error indicator (if it's negative).

```
public interface ArmTransaction extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int bindThread();
    public long block();
    public ArmApplication getApplication();
    public String getContextURIValue();
    public String getContextValue(int index);
    public ArmCorrelator getCorrelator();
    public ArmCorrelator getParentCorrelator();
    public int getStatus();
    public ArmTransactionDefinition getDefinition();
    public ArmUser getUser();
    public boolean isTraceRequested();
    public int reset();
    public int setArrivalTime();
    public int setContextURIValue(String value);
    public int setContextValue(int index, String value);
    public int setTraceRequested(boolean traceState);
    public int setUser(ArmUser user);
    public int start();
    public int start(byte[] parentCorr);
    public int start(byte[] parentCorr, int offset);
    public int start(ArmCorrelator parentCorr);
    public int stop(int status);
    public int stop(int status, String diagnosticDetail);
    public int unbindThread();
    public int unblock(long blockHandle);
    public int update();
}
```

9.43 org.opengroup.arm40.transaction.ArmTransactionDefinition

ArmTransactionDefinition contains the metadata that is the same for all instances of a transaction type (represented by ArmTransaction or ArmTranReport). It is created with the **newArmTransactionDefinition()** method of ArmTransactionFactory. ArmTransactionDefinition has the following attributes, all of which are immutable:

- The definition of the application that contains the transaction. It must not be null.
- The name of the transaction (maximum 127 characters).
- (optional) Identity property names and values, and context property names. They may be null.
- (optional) ID. An optional 16-byte ID may be associated with the identity of a transaction definition. The returned value, which could be null, is the same value passed to the **newArmTransactionDefinition()** method of ArmTransactionFactory. The ID value is bound to a unique combination of the application identity (represented by ArmApplicationDefinition), transaction name, any URI identity property, any identity property names and values, and any context property names. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

```
public interface ArmTransactionDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmApplicationDefinition getApplicationDefinition();
    public ArmID getID();
    public ArmIdentityPropertiesTransaction getIdentityProperties();
    public String getName();
}
```

9.44 org.opengroup.arm40.transaction.ArmTransactionFactory

The class that implements the ArmTransactionFactory interface is used to create all objects that implement interfaces in the org.opengroup.arm40.transaction package. Refer to the interface descriptions for an explanation of the parameters.

newArmApplication()
Creates the ArmApplication object to which transaction instances are related. definition is the only required non-null parameter.

newArmApplicationDefinition()
Creates the ArmApplicationDefinition object that describes the metadata about an application; that is, the descriptive data that is the same for all instances of the same application. name is the only required non-null parameter.

newArmCorrelator()
Creates a correlator from a byte array in the correct format. No length field is passed to **newArmCorrelator()** because ARM requires that the length of the correlator be found in the first two bytes (in network byte order) of the byte array (either at corrBytes or corrBytes+offset). The correlator must be no longer than the return value from the **getCorrelatorMaxLength()** method. If the correlator is longer than the ARM implementation supports, an ArmCorrelator object will be created, but it may contain dummy data, at the discretion of the ARM implementation.

newArmIdentityProperties()
Creates an object that contains an immutable set of identity property names and values, and an immutable set of context property names. These properties describe

identity properties that are common to all instances of an application or transaction. The names and values are provided in arrays of strings.

There can be up to twenty elements in each of the name and value arrays. The identity property name and value arrays should contain the same number of elements; for each non-null name or value, there should be a corresponding non-null value or name, respectively. For any array index, if either the name or the value is null or a zero-length string, both the name and the value are treated as being null. It is permissible to have null elements in the middle of the array. For example, it is permissible for the elements at indices 0 and 19 to be non-null, and all the elements from indices 1 to 18 to be null.

newArmTransaction()
Creates an object that represents an instance of a transaction.

newArmTransactionDefinition()
Creates an object that represents the metadata about a transaction.

newArmUser()
Creates an ArmUser object that represents the user who invoked (directly or indirectly) the transaction.

newArmID()
Creates the objects that contain an immutable 16-byte ID. Having 16-byte IDs accommodates some widely used IDs, such as the UUID (universally unique identifier) defined in the DCE and IETF standards.

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all zeros. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct; that is, any of its methods can be invoked without causing an exception, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 7 for a more complete explanation.

ArmTransactionFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(**null**)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(**null**)** is executed. For a broader discussion of error handling, refer to Chapter 7.

```
public interface ArmTransactionFactory extends ArmInterface {
// Public Constants
    public static final String propertyKey;
// Public Instance Methods
    public ArmApplication newArmApplication(
        ArmApplicationDefinition definition,
        String group,
        String instance,
        String[] contextValues);
    public ArmApplicationDefinition newArmApplicationDefinition(
        String name,
        ArmIdentityProperties identityProperties,
        ArmID id);
    public ArmCorrelator newArmCorrelator(
        byte[] corrBytes);
    public ArmCorrelator newArmCorrelator(
        byte[] corrBytes,
        int offset);
    public ArmID newArmID(
        byte[] idBytes);
    public ArmID newArmID(
        byte[] idBytes,
        int offset);
    public ArmIdentityProperties newArmIdentityProperties(
        String[] identityNames,
        String[] identityValues,
        String[] contextNames);
    public ArmIdentityPropertiesTransaction
        newArmIdentityPropertiesTransaction(
        String[] identityNames,
        String[] identityValues,
        String[] contextNames,
        String uriValue);
    public ArmTransaction newArmTransaction(
        ArmApplication app,
        ArmTransactionDefinition definition);
    public ArmTransactionDefinition newArmTransactionDefinition(
        ArmApplicationDefinition app,
        String name,
        ArmIdentityPropertiesTransaction identityProperties,
        ArmID id);
    public ArmUser newArmUser(
        String name,
        ArmID id);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

9.45 org.opengroup.arm40.metric.ArmTransactionWithMetrics

ArmTransactionWithMetrics is a subclass of ArmTransaction which is used if the application wishes to use metrics. All the ArmTransaction rules for using **start()**, **stop()**, etc., apply to ArmTransactionWithMetrics.

ArmTransactionWithMetrics extends ArmTransaction by adding methods to manipulate metrics. The ArmMetric subclass objects are bound to an ArmTransactionWithMetrics object when the ArmTransactionWithMetrics is created. This is done by specifying ArmMetricGroup in the **newArmTransactionWithMetrics()** method of ArmMetricFactory.

getTransactionWithMetricsDefinition()
Returns the ArmTransactionWithMetricsDefinition object that contains the metadata describing this transaction, including the metric definitions.

getMetricGroup()
Returns the ArmMetricGroup object that was bound when ArmTransactionWithMetrics is created. The returned value may be null.

```
public interface ArmTransactionWithMetrics extends ArmTransaction {
// No Public Constructors
// Public Instance Methods
    public ArmTransactionWithMetricsDefinition
        getTransactionWithMetricsDefinition();
    public ArmMetricGroup getMetricGroup();
}
```

9.46 org.opengroup.arm40.metric.ArmTransactionWithMetricsDefinition

ArmTransactionWithMetricsDefinition subclasses ArmTransactionDefinition to add a binding with an ArmMetricGroupDefinition. It contains the metadata that is the same for all ArmTransactionWithMetrics (or ArmTranReportWithMetrics) instances with the same identity.

The properties that are accessible via this interface are the same as ArmTransactionDefinition, plus the following:

- The metric group definition through which the metric definitions are known. The metric definitions contain the metadata about the metrics.

```
public interface ArmTransactionWithMetricsDefinition extends
    ArmTransactionDefinition {
// No Public Constructors
// Public Instance Methods (in addition to several inherited from
// ArmTranscationDefinition)
    public ArmMetricGroupDefinition getMetricGroupDefinition();
}
```

9.47 org.opengroup.arm40.transaction.ArmUser

ArmUser represents a user on behalf of whom a transaction is executed. It is created with the **newArmUser()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Name: The maximum length is 127 characters (CIM allows 256 but ARM 2.0 allows 128 bytes, including the null-termination character, so 127 is used). The name must not be null or zero length.

- (optional) A 16-byte ID is optionally associated with each ArmUser. It is provided by the application. If the value is null, no ID was provided.

```
public interface ArmUser extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmID getID();
    public String getName();
}
```

A Application Instrumentation Sample

```
/* ---------------------------------------------------------------- */
/* ------- ARM4 Java API Example: Automated Teller Machine ------- */
/* ---------------------------------------------------------------- */

/* ---------------------------------------------------------------- */
/* Example: ARMed Automated Teller Machine simulation               */
/*                                                                  */
/* Description:                                                     */
/*                                                                  */
/* This ATM example demonstrates a more complex ARMed application  */
/* using the new ARM4 Java API. It uses ARM correlators, ARM user, */
/* ARM metrics, diagnostic detail for error reporing, and the      */
/* arrival time feature.                                            */
/*                                                                  */
/* 1. Instantiate ARM factory objects, then create ARM definition  */
/*    objects for applications, transactions, or metrics using     */
/*    ARM factory methods.                                          */
/* 2. Create ARM object instaces using the definitions and         */
/*    factories.                                                    */
/* 3. Start and stop any transaction using the start() and stop()  */
/*    methods of the ArmTransactionWithMetrics object.             */
/* 4. Stop the ARM application instance using its end() method.    */
/* ---------------------------------------------------------------- */

import org.opengroup.arm40.transaction.*;
import org.opengroup.arm40.metric.*;

public class Atm implements ArmErrorCallback {

/* ---------------------------------------------------------------- */
/* -------------- ATM inner classes and constants ---------------- */
/* ---------------------------------------------------------------- */

class AtmCustomer {
    public int no;
    public String name;
    public int pin;
    public double balance;
    public String currency;

    public AtmCustomer(int no, String name, int pin, double balance,
        String currency) {
            this.no = no;
            this.name = name;
            this.pin = pin;
            this.balance = balance;
            this.currency = currency;
        }
    }

AtmCustomer[] customers = new AtmCustomer[] {
    new AtmCustomer(1, "Mueller", 1234, 150.0, "EUR"),
    new AtmCustomer(2, "Miller", 5678, 100.0, "GBP"),
    new AtmCustomer(3, "Meyer", 4711, 200.0, "USD")
};

static final int ATM_ERR_NONE        = 0;
static final int ATM_ERR_INVALID_NO  = 1;
static final int ATM_ERR_INVALID_PIN = 2;
static final int ATM_ERR_OVERDRAWN   = 3;
static final int ATM_ERR_MAX         = 4;

static final String[] errors = new String[] {
    "no error",
    "invalid customer number",
    "invalid pin number",
    "account overdrawn"
};

/* ------------------------------------------------------------- */
/* ---------------------- ARM objects --------------------------- */
/* ------------------------------------------------------------- */

// factories
ArmTransactionFactory tranFactory;
ArmMetricFactory metricFactory;

// application definitions
ArmApplicationDefinition appDef;

// transaction definitions
ArmTransactionDefinition atmTranDef;
ArmTransactionWithMetricsDefinition checkCustomerTranDef;
ArmTransactionWithMetricsDefinition withdrawTranDef;

// identity properties
ArmIdentityPropertiesTransaction withdrawIdProperties;

// metric definitions
ArmMetricNumericId32Definition customerNoMetDef;
ArmMetricNumericId32Definition customerPinMetDef;
ArmMetricGroupDefinition checkCustomerMetGroupDef;

ArmMetricCounter32Definition amountMetDef;
ArmMetricGauge32Definition oldBalanceMetDef;
ArmMetricGauge32Definition newBalanceMetDef;
ArmMetricGroupDefinition withdrawMetGroupDef;

// application instance
ArmApplication app;
```

```
// transaction instances
ArmTransaction atmTran;
ArmTransactionWithMetrics checkCustomerTran;
ArmTransactionWithMetrics withdrawTran;

// metric objects
ArmMetricNumericId32 customerNoMet;
ArmMetricNumericId32 customerPinMet;
ArmMetricGroup checkCustomerMetGroup;

ArmMetricCounter32 amountMet;
ArmMetricGauge32 oldBalanceMet;
ArmMetricGauge32 newBalanceMet;
ArmMetricGroup withdrawMetGroup;

// correlators for parent transactions */
ArmCorrelator atmCorr;
ArmCorrelator checkCustomerCorr;

java.io.BufferedReader inputReader = new java.io.BufferedReader(
    new java.io.InputStreamReader(System.in));

/* ------------------------------------------------------------ */
/* -------------------- ARM initialization ---------------------- */
/* ------------------------------------------------------------ */

void initPropertiesForARMSdk() {
    java.util.Properties p = System.getProperties();

    p.setProperty(ArmTransactionFactory.propertyKey,
    "org.opengroup.arm40.sdk.ArmTransactionFactoryImpl");

    p.setProperty(ArmMetricFactory.propertyKey,
    "org.opengroup.arm40.sdk.ArmMetricFactoryImpl");
}

ArmTransactionFactory getTranFactory() {
    java.util.Properties p = System.getProperties();
    String tranFactoryName =
        p.getProperty(ArmTransactionFactory.propertyKey);
    if (tranFactoryName == null) {
        System.err.println("Could't getProperty " +
        ArmTransactionFactory.propertyKey);
        return null;
    }

    Class tranFactoryClass;
    ArmTransactionFactory tranFactory;

    try {
        tranFactoryClass = Class.forName(tranFactoryName);
    }
    catch (ClassNotFoundException e) {
        System.err.println("Could't find class " + tranFactoryName);
```

```
            return null;
        }

        try {
        tranFactory =
            (ArmTransactionFactory)tranFactoryClass.newInstance();
        }
        catch (Exception e) {
            System.err.println("Couldn't instantiate " +
                tranFactoryName + ":\n" + e.getMessage());
            return null;
        }

        tranFactory.setErrorCallback(this);

        return tranFactory;
    }

    ArmMetricFactory getMetricFactory() {
        java.util.Properties p = System.getProperties();
        String metFactoryName =
            p.getProperty(ArmMetricFactory.propertyKey);

        if (metFactoryName == null) {
            System.err.println("Could't getProperty " +
                ArmMetricFactory.propertyKey);
        return null;
        }

        Class metFactoryClass;
        ArmMetricFactory metFactory;

        try {
            metFactoryClass = Class.forName(metFactoryName);
        }
        catch (ClassNotFoundException e) {
            System.err.println("Could't find class " + metFactoryName);
            return null;
        }

        try {
            metFactory = (ArmMetricFactory) metFactoryClass.newInstance();
        }
        catch (Exception e) {
            System.err.println("Couldn't instantiate " +
                metFactoryName + ":\n" + e.getMessage());
            return null;
        }

        metFactory.setErrorCallback(this);

        return metFactory;
    }
```

```
void initFactories() throws Exception {
    tranFactory = getTranFactory();
    metricFactory = getMetricFactory();
}

void armInit() {
    // create application definition
    appDef = tranFactory.newArmApplicationDefinition(
        "ATM - Automated Teller Machine", null, null);

    // create metric definitions
    customerNoMetDef = metricFactory.newArmMetricNumericId32Definition(
        appDef, "Customer-No.", "ID",
        ArmMetricDefinition.METRIC_USE_GENERAL, null);
    customerPinMetDef =
        metricFactory.newArmMetricNumericId32Definition(
        appDef, "Customer-Pin", "ID",
        ArmMetricDefinition.METRIC_USE_GENERAL, null);

    checkCustomerMetGroupDef =
        metricFactory.newArmMetricGroupDefinition(
        new ArmMetricDefinition[]
            {customerNoMetDef,customerPinMetDef});

    amountMetDef = metricFactory.newArmMetricCounter32Definition(
        appDef, "Amount", "money",
        ArmMetricDefinition.METRIC_USE_GENERAL, null);
    oldBalanceMetDef = metricFactory.newArmMetricGauge32Definition(
        appDef, "Old Balance", "money",
        ArmMetricDefinition.METRIC_USE_GENERAL, null);
    newBalanceMetDef = metricFactory.newArmMetricGauge32Definition(
        appDef, "New Balance", "money",
        ArmMetricDefinition.METRIC_USE_GENERAL, null);

    withdrawMetGroupDef = metricFactory.newArmMetricGroupDefinition(
        new ArmMetricDefinition[] {amountMetDef,oldBalanceMetDef,
            newBalanceMetDef});

    // create transaction definitions
    atmTranDef = tranFactory.newArmTransactionDefinition(
        appDef, "ATMTran", null, null);

    checkCustomerTranDef = metricFactory.
        newArmTransactionWithMetricsDefinition(
            appDef, "Check Customer", null,
                checkCustomerMetGroupDef, null);

// The withdraw transaction uses a "Currency" context value, so we
// need to set up a properties object.
    withdrawIdProperties =
        tranFactory.newArmIdentityPropertiesTransaction(
        null, null, new String[] {"Currency"}, null);
    withdrawTranDef =
        metricFactory.newArmTransactionWithMetricsDefinition(
            appDef, "Withdraw", withdrawIdProperties,
            withdrawMetGroupDef, null);

// Create the application instance object.
    app = tranFactory.newArmApplication(appDef, "Examples",
        null, null);

// Create transaction instance objects.
    atmTran = tranFactory.newArmTransaction(app, atmTranDef);

// For transaction instance objects with metrics, metric instances
// and groups must be created in advance.
    customerNoMet =
        metricFactory.newArmMetricNumericId32(customerNoMetDef);
    customerPinMet =
        metricFactory.newArmMetricNumericId32(customerPinMetDef);
    checkCustomerMetGroup = metricFactory.newArmMetricGroup(
        checkCustomerMetGroupDef, new ArmMetric[] {
            customerNoMet, customerPinMet});

    checkCustomerTran = metricFactory.newArmTransactionWithMetrics(
        app, checkCustomerTranDef, checkCustomerMetGroup);

    amountMet = metricFactory.newArmMetricCounter32(amountMetDef);
    oldBalanceMet =
        metricFactory.newArmMetricGauge32(oldBalanceMetDef);
    newBalanceMet =
        metricFactory.newArmMetricGauge32(newBalanceMetDef);
    withdrawMetGroup = metricFactory.newArmMetricGroup(
        withdrawMetGroupDef, new ArmMetric[] {
            amountMet, oldBalanceMet, newBalanceMet});

    withdrawTran = metricFactory.newArmTransactionWithMetrics(
        app, withdrawTranDef, withdrawMetGroup);
}

void free() {
    // Stop the application instance.
    app.end();

    // Destroy all registered metadata.
    appDef.destroy();
}

/* ---------------------------------------------------------------- */
/* - method implementation satisfying interface ArmErrorCallback -- */
/* ---------------------------------------------------------------- */

public void errorCodeSet(ArmInterface errorObject,
    String interfaceName, String methodName) {
        System.err.println("*** Error callback: ***");
        System.err.println(" Class: " +
                errorObject.getClass().getName());
            System.err.println(" Interface: " + interfaceName);
            System.err.println(" Method: " + methodName);
            System.err.println(" Description: " +
                errorObject.getErrorMessage(errorObject.getErrorCode()) +
                " (" +   errorObject.getErrorCode() + ")");
            System.err.println("**********************");
    }

    void printError(int err) {
        if (err < errors.length)
            System.err.println("\nerror: " + errors[err]);
        else
            System.err.println("\nnknown nerror");
    }

    /* ---------------------------------------------------------------- */
    /* ------------ The actual ATM worker methods --------------------- */
    /* ---------------------------------------------------------------- */

    void checkCustomer() {
        long arrivalTime;

        int idx=-1;
        int no;
        int pin;

        int err = ATM_ERR_INVALID_NO;

        /* The real check_customer transaction starts here. But currently
        /* we don't know the customer (arm user) so get a time stamp from
        /* ARM and set it as actual arrival time that is used for the
        /* call of the start() method later on. */
        checkCustomerTran.setArrivalTime();

        try {
            System.out.print("enter your customer no.: ");
            no = Integer.parseInt(inputReader.readLine());

            System.out.print("enter your pin no.: ");
            pin = Integer.parseInt(inputReader.readLine());
        }
        catch (Exception e) {
            System.out.println("an error occured during entry: " +
                e.getMessage());
            return ;
        }

        for(int i=0; i<customers.length; ++i) {
            if(customers[i].no == no) {
                if(customers[i].pin == pin) {
                    idx = i;
                    err = ATM_ERR_NONE;
                }
            else err = ATM_ERR_INVALID_PIN;

            break;
            }
        }

        if(idx >= 0) {
            ArmUser currentUser =
                tranFactory.newArmUser(customers[idx].name, null);
            checkCustomerTran.setUser(currentUser);
        }

        customerNoMet.set(no);
        customerPinMet.set(pin);

        // Now start the check_customer transaction.
        checkCustomerTran.start(atmCorr);

        checkCustomerCorr = checkCustomerTran.getCorrelator();

        if(idx >= 0)
            withdraw(idx);

        if(err > ATM_ERR_NONE) {
            checkCustomerTran.stop(ArmConstants.STATUS_FAILED,
                errors[err]);
            printError(err);
        }
        else
            checkCustomerTran.stop(ArmConstants.STATUS_GOOD);
    }

    void withdraw(int idx) {
        int err = ATM_ERR_NONE;
        double amount = 0.0;
        double newBalance;
        double oldBalance;
        String currency;

        oldBalance = newBalance = customers[idx].balance;

        // We want the metric values to appear only at stop time,
        // so disable them for now.
        withdrawMetGroup.setMetricValid(0, false);
        withdrawMetGroup.setMetricValid(1, false);
        withdrawMetGroup.setMetricValid(2, false);

        withdrawTran.setContextValue(0, customers[idx].currency);

        withdrawTran.start(checkCustomerCorr);

        System.out.println(" Hello Mr./Mrs./Miss " + customers[idx].name);
        System.out.println(" your balance is " + customers[idx].balance +
            " " + customers[idx].currency);
        try {
            System.out.print(" enter amount to withdraw in " +
                customers[idx].currency + " :");
            amount = Double.parseDouble(inputReader.readLine());

            if(amount > oldBalance)
                err = ATM_ERR_OVERDRAWN;
            else
                newBalance = oldBalance - amount;

            amountMet.set((int)(amount * 100.0));
            oldBalanceMet.set((int)(oldBalance * 100.0));
            newBalanceMet.set((int)(newBalance * 100.0));
        }
        catch (Exception e) {
            System.out.println("an error occured during entry: " +
                e.getMessage());
            amount = 0.0;
            currency = "none";
        }

        amountMet.set((int)(amount * 100.0));
        oldBalanceMet.set((int)(oldBalance * 100.0));
        newBalanceMet.set((int)(newBalance * 100.0));

        // Re-enable the recording of the metric values.
        withdrawMetGroup.setMetricValid(0, true);
        withdrawMetGroup.setMetricValid(1, true);
        withdrawMetGroup.setMetricValid(2, true);

        if(err > ATM_ERR_NONE) {
            withdrawTran.stop(ArmConstants.STATUS_FAILED, errors[err]);
            printError(err);
        }
        else {
            System.out.println(" your new balance is " + newBalance);
            withdrawTran.stop(ArmConstants.STATUS_GOOD);
        }
    }

    void run() {
        // Now start the transaction.
        atmTran.start();
        atmCorr = atmTran.getCorrelator();

        System.out.println("atm:");
        System.out.println("----");

        checkCustomer();

        System.out.println("bye.");

        // Stop the measurement for the atm transaction and commit
        // it to ARM.
        atmTran.stop(ArmConstants.STATUS_GOOD);
    }

    public static void main(String args[]) {
        Atm atm = new Atm();

        atm.initPropertiesForARMSdk();
        try {
            atm.initFactories();
        }
        catch (Exception e) {
            System.err.println("Exception during factory init");
            System.err.println("This should not happen with "+
                "a well-behaved ARM implementation");
            System.exit(-1);
        }

        atm.armInit();

        atm.run();

        atm.free();
    }
    }
```

B Information for Implementers

This appendix contains information useful to creators of ARM implementations, and analysis and reporting programs that process ARM data. Applications using ARM to measure transactions do not use any of this information.

Byte Ordering in Correlators

Correlators are passed from application to application. The transfer may occur within a single system or a single JVM (Java Virtual Machine), or it may occur across a network. The recipient and sender of a correlator may run on different machines with different architectures, and the conventions for ordering bytes in data fields, such as integers and arrays, may be different.

If all the programs that touch a correlator are written in Java, the JVM would ensure that the same ordering conventions are followed and no order would need to be specified. However, correlators are meant to be passed between applications using any version of ARM (both C and Java) and running on any platform, including both big-endian and little-endian platforms. Because big-endian and little-endian platforms order bytes differently, the specification needs to explicitly state the required ordering, in order to make the correlators interchangeable.

Recognizing this fact, ARM is designed expressly to permit correlators to be exchanged between any application using ARM and any ARM implementation, regardless of how it is written. For example, an application using ARM 4.0 Java Bindings may receive a (parent) correlator from an application using ARM 2.0 (for C programs), and it may send its correlator to an application using ARM 3.0 for Java programs. To permit these types of exchanges, ARM specifies the ordering of bytes within the correlator.

All correlator fields, and the correlator itself, are sent in network byte order. Network byte order is a standard described as follows. The most significant bit is the first bit sent, and the least significant bit is the last bit sent. For example, a 32-bit integer field would be sent with the most significant byte first, and the least significant byte would be the fourth byte sent.

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Bit 0 is the most significant bit			Bit 31 is the least significant bit

Limits on Interoperability between ARM Implementations

There is one limit on interoperability. In ARM 2.0 and 3.0, the maximum length of a correlator is 168 bytes. ARM 4.0 has changed the maximum to 512 bytes. If ARM 4.0 implementations restrict themselves to correlators of no more than 168 bytes, then the correlators are fully interchangeable with any other version of ARM. If an ARM 4.0 implementation uses a correlator that is more than 168 bytes long, it can only be successfully interchanged with another ARM 4.0 implementation.

Correlator Formats

ARM specifies formatting constraints that all correlators must adhere to. These are described in the following section. In addition, different versions of ARM have defined three specific correlator formats. ARM 4.0 has not defined any formats, and, in general, has taken the approach of making correlators as opaque as possible.

ARM Correlator Format Constraints

These constraints apply to all formats.

Table 1: ARM Correlator Format Constraints

Position	**Length**	**Contents**
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes. Valid lengths are 4 <= length <= 512. Lengths shorter than four bytes are not permitted because all correlators must have the four bytes defined in this table. Note that a correlator that is longer than 168 bytes could not be passed to and used by an application using ARM 2.0 or ARM 3.0, because the maximum size in those versions was 168 bytes. Some correlator formats impose shorter length restrictions. In particular, formats 1, 2, and 127 have a maximum of 168 bytes.
Byte 2	1 byte	Correlator format The range 0:127 (unsigned) is reserved by the ARM specification. Six values have been assigned: 1 – Defined in ARM 2.0 2 – Defined in ARM 3.0 28 – Reserved for Hewlett-Packard 103 – Reserved for IBM 122 – Reserved for tang-IT 127 – Defined in ARM 3.0 The range 128:255 (unsigned) is available for use by ARM implementers. Known used values include: 128 – Hewlett-Packard 203 – IBM 204 – IBM
Byte 3	1 byte	Flags All eight-bit flags are reserved by the ARM specification. Two flags are defined in positions 0:1 (the highest order bits), as in `ab000000`, where `a` and `b` are bit flags.

Position	Length	Contents
		a = 1 if a trace of this transaction is requested by the agent that generated the correlator. This is transparent to the applications. b = 1 if the application indicates that this transaction is of particular importance, such as a test transaction, and therefore worthy of being traced. There are no requirements for how these flags are handled, if at all. By convention, if the flags are turned on in a correlator, the setting is copied into the correlators for child transactions. However, local policy or the ARM implementation may override this convention. The usage scenario that led to their creation was to enable a trace of selected transactions throughout an enterprise. A selective trace would yield much useful information without being a significant burden on the systems processing the transaction. For example, a client could be experiencing response time problems. The agent on the client could turn on the trace flag (bit 0) in the correlators that it generates. When this correlator is passed, as the parent correlator, to the ARM implementation on the server, the ARM implementation could turn on the trace flag in the correlators that it generates. The process could continue recursively. What has resulted is a trace of all the transactions associated with the client experiencing the response time problem, but only those transactions. If there are 1,000 clients in the enterprise running this application, 0.1% of all transactions are traced, which is a minimal load on the systems. The value of a surgical trace like this was considered great enough to justify including it in the ARM specification.

Index

applet 24
application instrumentation 70
ARM
    compatibility between versions....4
    evolution of 3
    usage of 1
ARM 4.0
    new capabilities............................3
ARM objects 20, 21, 24
arm.properties.................................24
ArmApplication..........................29, 35
ArmApplicationDefinition 29, 36
ArmApplicationRemote...................37
ArmConstants 37
ArmCorrelator 38
ArmErrorCallback 39
ArmID...40
ArmIdentityProperties.....................40
ArmInterface 42
ArmMetric 42
ArmMetricCounter32..........................43
ArmMetricCounter32Definition 43
ArmMetricCounter64 43
ArmMetricCounter64Definition 44
ArmMetricCounterFloat32 44
ArmMetricCounterFloat32Definition44
ArmMetricDefinition 44
ArmMetricFactory 45
ArmMetricGauge32 50
ArmMetricGauge32Definition 50
ArmMetricGauge64 50
ArmMetricGauge64Definition 50
ArmMetricGaugeFloat32 51
ArmMetricGaugeFloat32Definition .51
ArmMetricGroup 51
ArmMetricGroupDefinition 52
ArmMetricNumericId32...................52
ArmMetricNumericId32Definition....53
ArmMetricNumericId64...................53
ArmMetricNumericId64Definition....53
ArmMetricString32..........................53
ArmMetricString32Definition...........54
ArmSystemAddress........................54
ArmToken 55
ArmTranReport....................6, 31, 56
ArmTranReportFactory...................59
ArmTranReportWithMetrics 17, 60
ArmTransaction 6, 8, 29, 61
ArmTransactionDefinition 29, 64
ArmTransactionFactory 65
ArmTransactionWithMetrics ... 16, 67
ArmTransactionWithMetricsDefinition
   68
ArmUser 68
calling hierarchy.............................13
class loaders 23
context properties...........................40
convenience methods 23
correlator 12
    byte ordering.............................80
correlator formats 81
    constraints.................................81
counters 15
data categories...............................14
data model 29
distributed transaction 11
error handling 26
factory class...................................22
factory interfaces............................22
gauges 15
    conventions...............................18
heartbeat..8
identification information 14
identity properties 40
interoperability limits 80
Java Binding.....................................6
Java Bindings 4
Java interfaces 20
Java packages 4
java.lang.System 22
java.util.Properties..........................22
JNI 23
JVM.........................4, 10, 22, 57, 80
management agent...........................2

Management Agent 4
measure transactions.........................8
measurement information...............14
method
    naming conventions 34
metrics 14, 29
    multiple values 17
numeric IDs 15
org.opengroup.arm40.* packages...32
programming options 8
property keys.................................22
propertyKey 22
response time.................................14
status 14
stop time..14
strings 16
transaction measurement..................2
transaction relationships.................11