esoc

European Space Operations Centre
Robert-Bosch-Strasse 5
D-64293 Darmstadt
Germany
T +49 (0)6151 900
F +49 (0)6151 90495
www.esa.int

TECHNICAL NOTE

Cross Support Transfer Service API - Developer's Guide

Prepared by **Taylan Özden, Matteo Renesto**
Telespazio Germany GmbH

Reference
Issue/Revision **2.0**
Date of Issue **25/03/2022**
Status **Draft**

APPROVAL

Title Cross Support Transfer Service API - Developer's Guide	
Issue Number 2	**Revision Number** 1
Author Taylan Özden	**Date** 25/03/2022
Approved By	**Date of Approval**

CHANGE LOG

Reason for change	Issue Nr.	Revision Number	Date
Initial Delivery	1	0	28/03/2019
Changes on Documentation of Interfaces	1	1	17/05/2019
Changes on Documentation of Interfaces	1	2	12/07/2019

Table of contents:

1 INTRODUCTION 6
1.1 Purpose and Scope 6
1.2 Document Overview 6
2 APPLICABLE AND REFERENCE DOCUMENTS 7
2.1 Applicable Documents 7
2.2 Reference Documents 7
3 USAGE OF THE CSTS API 8
3.1 Importing the CSTS API 8
3.1.1 Maven Source Import 8
3.1.2 JAR File Import 9
3.2 Service Application 9
3.2.1 Predefined Service Application(s) 9
3.2.2 Implementing a Service Application 11
3.3 Service Instance 13
3.3.1 External Parameters 13
3.3.2 External Events 13
3.3.3 Production Status Change 14
3.3.4 Production Configuration Change 14
3.4 Procedures 14
3.4.1 IAssociationControl Interface 14
3.4.2 IUnbufferedDataDelivery Interface 15
3.4.3 IBufferedDataDelivery Interface 15
3.4.4 IBufferedDataProcessing Interface 15
3.4.5 ISequenceControlledDataProcessing Interface 15
3.4.6 IInformationQuery 16
3.4.7 ICyclicReport Interface 16
3.4.8 INotification Interface 16
3.4.9 IThrowEvent Interface 16
4 ARCHITECTURAL DESIGN 17
4.1 CSTS API Instances 17
4.2 Procedures 17
5 WORKFLOWS WITHIN THE CSTS API 19
5.1 State Machine 19
5.2 Parameters 20
5.3 Events 20
6 EXTENDING THE CSTS API 22
6.1 Suggested Approach 22
6.2 Essential Methods 23
6.2.1 Initialization of Operation Types 23
6.2.2 Type and Version 23
6.2.3 Termination of Procedures 23
6.2.4 Creation of Operations 23
6.2.5 Initiation and Information of passed Operations 24
6.2.6 Encoding and Decoding Operations 24
6.3 Diagnostics 25

6.4 Procedure States 25
6.5 Parameters 26
6.6 Procedure Configuration Parameters 27
6.7 Procedure Events 28
6.8 New Standards 29
7 PREDEFINED CCSDS TYPE CLASSES 29
7.1 Object Identifier 29
7.2 Name 30
7.3 Label 30
7.4 Label List 30
7.5 Qualified Parameters 31
7.6 Time 34
7.7 Conditional Time 34
7.8 Duration 34
7.9 Extension 35

Table of figures:

Figure 1: CSTS API Source Import 8

Figure 2: CSTS API JAR File Import 9

Figure 3: CSTS API Architecture 17

Figure 4: Procedure Architecture 18

Figure 5: State Machine 19

Figure 6: CSTS Parameters 20

Figure 7: CSTS Events 21

Figure 8: Suggested Approach for extending Procedures 22

Figure 9: Architecture of Parameters 27

Figure 10: Qualified Parameters Mapping 31

Figure 11: Parameter Type Mapping 33

Table of tables:

NO TABLE OF FIGURES ENTRIES FOUND.

1 INTRODUCTION

The Cross Support Transfer Service (CSTS) API is a Java 8 framework implementing the defined standards of the CCSDS Blue Book *Cross Support Transfer Service – Specification Framework* (CCSDS 921.1-B-1).

1.1 Purpose and Scope

This document is the Developer's Guide for version 1.0 of the CSTS API. It provides necessary information on how to use the CSTS API to build new services as well as the extension of CSTS framework procedures to add new procedures conforming the provided API.

1.2 Document Overview

The Developer's Guide is structured as follows:

- Section 2 lists applicable and referenced documents
- Section 3 introduces the usage of the CSTS API by building a service based upon provided and extended procedures
- Section 4 outlines the architectural design to provide an overview of necessary information relevant while extending the CSTS framework
- Section 5 describes and visualizes workflows within the CSTS API
- Section 6 provides information on how the extend the CSTS framework by new procedures
- Section 6.8 lists predefined CCSDS types and how they are being used in their Java context within the CSTS API

2 APPLICABLE AND REFERENCE DOCUMENTS

This section lists applicable and referenced documents with their issue, revision and date.

2.1 Applicable Documents

Ref.	Document Title	Issue and Revision, Date
[AD-1]	Cross Support Transfer Service – Specification Framework (CCSDS 921.1-B-1)	Issue 1, April 2017

2.2 Reference Documents

Ref.	Document Title	Issue and Revision, Date
[RD-1]	Cross Support Transfer Service – Configuration Files	1.0, 01/04/2019

3 USAGE OF THE CSTS API

This section introduces the usage of the CSTS API by building a service based on provided procedures as well as procedures, which were created by extending existing procedures as suggested as in Section 6.

3.1 Importing the CSTS API

3.1.1 Maven Source Import

The CSTS API source is delivered as a Maven project. The import functionality for Maven projects of Eclipse is sufficient to import the CSTS API and resolve all dependencies.

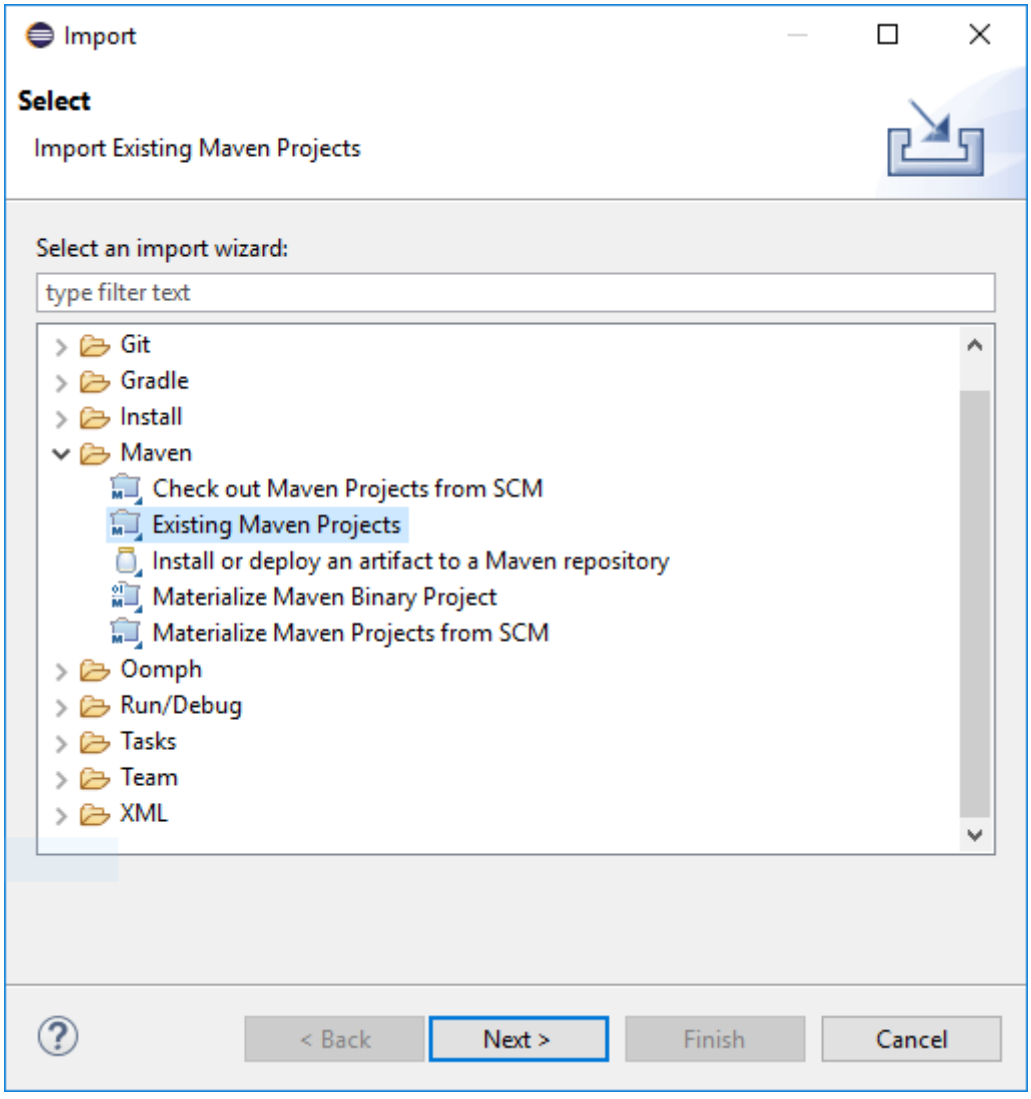

Figure 1: CSTS API Source Import

3.1.2 *JAR File Import*

The CSTS API is also delivered as a single JAR file. The attachment of the JAR file to the build path issufficient to import the CSTS API and resolve all dependencies.

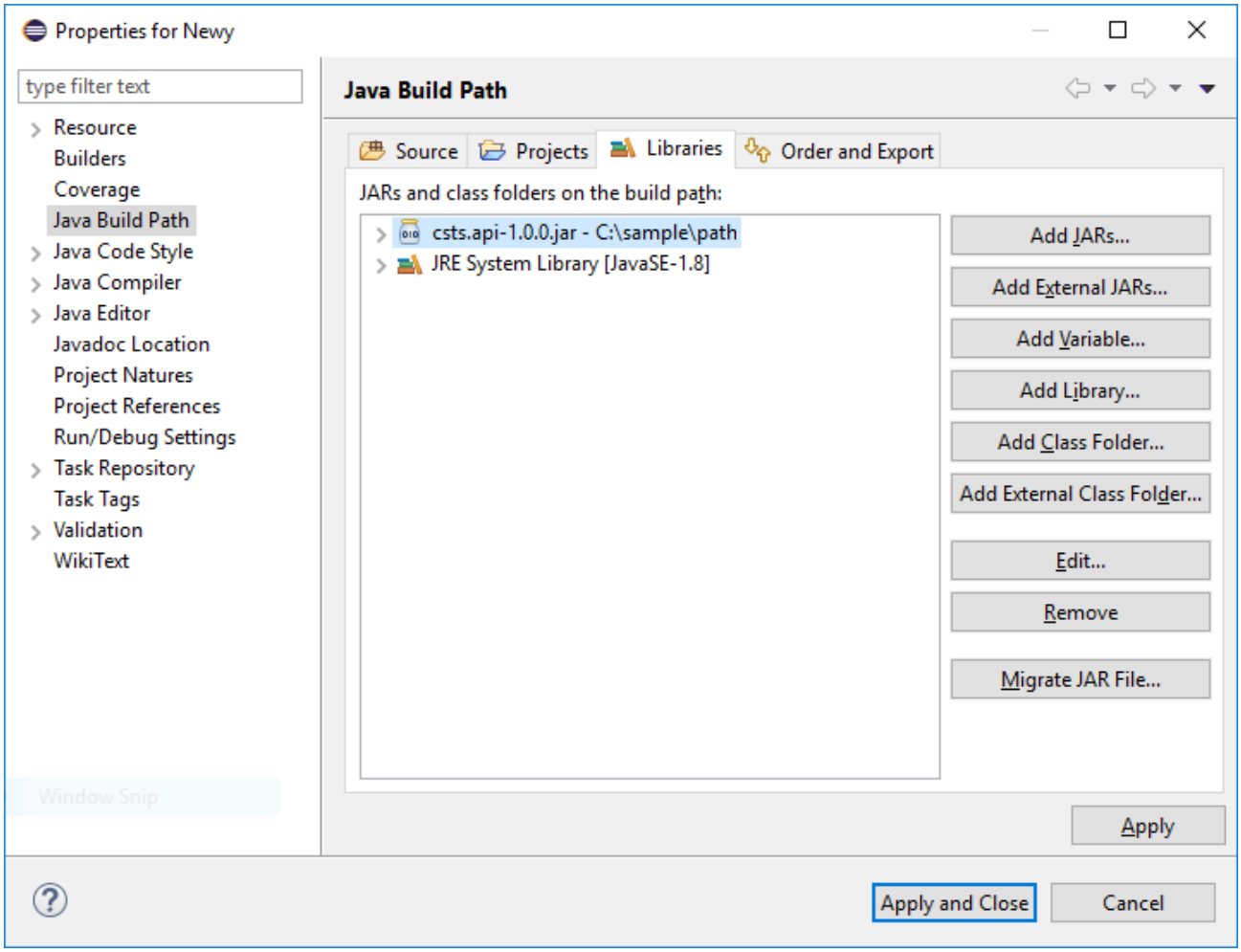

Figure 2: CSTS API JAR File Import

3.2 Service Application

In order to be able to use the CSTS API, it is necessary to have an appropriate Service Application. Some service applications are already predefined and offered as part of the API library itself; however, it is also possible to develop new service applications to serve specific needs. The next two subsections focus respectively on these two cases.

3.2.1 *Predefined Service Application(s)*

The current version of the CSTS API library comes with predefined Service Application for the **Monitoring Data services** (CyclicReport, OnDemandCyclicReport, Notification, and

Query), and for the **Return Cfdp PDU service**. They can be found respectively under the following packages:

esa.egos.csts.app.si.md

esa.egos.csts.app.si.rtn.cfdp.pdu

Sample snippet of Monitoring Data Provider:

```
IMonitoringDataSiProvider providerSi = MonitoringDataSiProvider.builder(providerApi, providerConfig)
  .addCyclicReportProcedure(procedureConfig, 0)//add one cyclic report procedure
  .addOnChangeCylicReportProcedure(procedureConfig, 0)
  .build();

//Create your parameters and add them to the provider (must be done before user starts)
IParameter parameter1 = createFunctionalResourceParameter(125);
IParameter parameter2 = createFunctionalResourceParameter(456);
providerSi.addParameters(Arrays.asList(parameter1,parameter2));
//set new values if desired (function not in API)
setFunctionalResourceValue(parameter1, 127);
```

Sample snippet of Monitoring Data User:

```
IMonitoringDataSiUser userSi = MonitoringDataSiUser.builder(userApi, userConfig, serviceVersion)
    .addCyclicReportProcedure(procedureConfig, 0) // add one cyclic report procedure
    .addOnChangeCylicReportProcedure(procedureConfig, 0)
    .setParameterListener((pii, parameters) -> {
      //write here your callback
      })
      .build();
userSi.bind();
```

```
// Crete identifier for the two procedures
ProcedureInstanceIdentifier cyclicReportInstance =

ProcedureInstanceIdentifier.of(ProcedureType.of(OIDs.cyclicReport),ProcedureRole.PRIME,
0);

ProcedureInstanceIdentifier onChangecyclicReportInstance =
      ProcedureInstanceIdentifier.of(ProcedureType.of(OIDs.ocoCyclicReport),ProcedureRo
le.SECONDARY,0);

userSi.startCyclicReport(cyclicReportInstance, cycleTime);
userSi.startCyclicReport(onChangecyclicReportInstance, cycleTime ,true);
//when shutting down
userSi.stopCyclicReport(cyclicReportInstance);
userSi.stopCyclicReport(onChangecyclicReportInstance);
userSi.unbind();
```

Note: gray items are not provided by the library, they must be created/initialized programmatically.

3.2.2 Implementing a Service Application

Applications using the CSTS API need to implement the **IServiceInform** interface. This interface informs the service about incoming operations. The operations are forwarded to the application through the following three methods depending on their type:

```
void informOpInvocation(IOperation operation)
void informOpAcknowledgement(IAcknowledgedOperation operation)
void informOpReturn(IConfirmedOperation operation)
```

The actual type of the forwarded operation is queried by the

`OperationType getType()`

method of the **IOperation** interface. The returned enumeration type contains all available CSTS operation types, as well as the forward buffer and return buffer types relevant to the **Buffered Data Delivery** and the **Buffered Data Processing** procedures.

Following sample demonstrates the implementation of a sample service built upon the **Information Query**, **Cyclic Report** and **Notification** procedures:

```
ICstsApi api;
IServiceInstance serviceInstance;
IInformationQuery informationQuery;
ICyclicReport cyclicReport;
INotification notification;

api = new CstsApi("Test Service", AppRole.PROVIDER);
api.initialize(configFile);

ObjectIdentifier spacecraftId = ObjectIdentifier.of(1,3,112,4,7,0);
ObjectIdentifier facilityId = ObjectIdentifier.of(1,3,112,4,6,0);
ObjectIdentifier typeId = ObjectIdentifier.of(1,3,112,4,4,1,2);
int instanceNumber = 0;

IServiceInstanceIdentifier identifier = new ServiceInstanceIdentifier(spacecraftId,
facilityId, typeId, instanceNumber);
serviceInstance = api.createServiceInstance(identifier, this);

informationQuery = serviceInstance.createProcedure(InformationQueryProvider.class);
informationQuery.setRole(ProcedureRole.PRIME, 0);
serviceInstance.addProcedure(informationQuery);

cyclicReport = serviceInstance.createProcedure(CyclicReportProvider.class);
cyclicReport.setRole(ProcedureRole.SECONDARY, 0);
serviceInstance.addProcedure(cyclicReport);
cyclicReport.getMinimumAllowedDeliveryCycle().initializeValue(50);

notification = serviceInstance.createProcedure(NotificationProvider.class);
notification.setRole(ProcedureRole.SECONDARY, 0);
serviceInstance.addProcedure(notification);

// the application needs to make sure that it chooses valid values from the proxy
configuration
serviceInstance.setPeerIdentifier(serviceInstance.getApi().getProxySettings().getRemote
PeerList().get(0).getId());
serviceInstance.setResponderPortIdentifier(serviceInstance.getApi().getProxySettings().
getLogicalPortList().get(0).getName());


serviceInstance.configure();
```

The configuration file parameter which is passed to the CSTS API is the specified path of the XML file provided by the application to configure the provider or user and its underlying communications service.

The example above can be analogously used for user instances. It is sufficient to exchange the AppRole by the user indicator and instantiate user classes of procedures while creating them. Additionally configuration parameters (if the procedure has any) does not need to be initialized for user instances. Configuration parameters only need to be initialized in provider instances and only in the case if they are not initialized by default.

In the provider example above the **Cyclic Report** procedure has two configuration parameters of which one need to be initialized. The initialization needs to take place after adding the procedure to the service instance. The second configuration parameter does not need to be initialized since its type is a list of labels, which is initialized by default and is empty at initialization.

3.3 Service Instance

3.3.1 External Parameters

The service instance provides a possibility to add external parameters (e.g. parameters of the implemented service) to be immediately available to all procedures requesting these parameters (see 5.2 and 6.5).

External parameters can be added or removed through following methods:

```
void addExternalParameter(IParameter parameter)
```

```
void removeExternalParameter(IParameter parameter)
```

3.3.2 External Events

The service instance also provides the possibility to add external events (e.g. events of the implemented service) which are immediately available to all procedures to subscribe to.

External events can be added or removed through following methods:

```
void addExternalEvent(IEvent event)
```

```
void removeExternalEvent(IEvent event)
```

3.3.3 Production Status Change

If the production status has changed, the **IServiceInstance** interface provides following method to change the current state:

```
void changeProductionState(ProductionState state)
```

The CSTS API also takes care of every additional measures to fire an Event (see 5.3) with the corresponding object identifier (see 7.1)

```
OIDs.svcProductionStatusChangeVersion1
```

to notify all subscribed procedures to use the incoming Event for further actions (such as the **Notification** procedure to notify a subscribed user service).

The value of the event is automatically set to a valid value and its time is being set to the current time in UTC.

3.3.4 Production Configuration Change

In case the production configuration has been changed, it is suggested to call the

```
void changeProductionConfiguration()
```

method of the **IServiceInstance** interface. Similar to 3.3.3, this method takes care of notifying subscribed procedures with the object identifier

```
OIDs.svcProductionStatusChangeVersion1
```

and sets the value of the event to empty. The time being set is the current time in UTC.

3.4 Procedures

All procedures provide helper methods which automatically take care of the creation of operations and requires arguments (if necessary) to correctly fill the invocation arguments.

These methods are the suggested method of sending operations using the CSTS framework to not breach any pattern and workflows within the CSTS API and its underlying state machine (see 5.1).

3.4.1 IAssociationControl Interface

```
CstsResult bind()
```

```
CstsResult unbind()
```

```
CstsResult abort(PeerAbortDiagnostics diagnostics)
```

3.4.2 IUnbufferedDataDelivery Interface

```
CstsResult requestDataDelivery()

CstsResult endDataDelivery()

CstsResult transferData(byte[] data)

CstsResult transferData(EmbeddedData embeddedData)
```

3.4.3 IBufferedDataDelivery Interface

```
CstsResult requestDataDelivery()

CstsResult requestDataDelivery(Time startGenerationTime, Time stopGenerationTime)

CstsResult endDataDelivery()

CstsResult transferData(byte[] data)

CstsResult transferData(EmbeddedData embeddedData)
```

3.4.4 IBufferedDataProcessing Interface

```
CstsResult requestDataProcessing()

CstsResult endDataProcessing()

CstsResult processData(long dataUnitId, byte[] data, boolean produceReport)

CstsResult processData(long dataUnitId, EmbeddedData embeddedData, boolean produceReport)

CstsResult processBuffer(List<Long> dataUnitIds, List<byte[]> data, List<Boolean>
produceReports)

CstsResult processEmbeddedBuffer(List<Long> dataUnitIds, List<EmbeddedData>
embeddedData, List<Boolean> produceReports)
```

3.4.5 ISequenceControlledDataProcessing Interface

```
CstsResult requestDataProcessing(long firstDataUnitId)

CstsResult endDataProcessing()

CstsResult processData(long dataUnitId, byte[] data, boolean produceReport)

CstsResult processData(long dataUnitId, byte[] data, Time earliestDataProcessingTime,
Time latestDataProcessingTime, boolean produceReport)
```

```
CstsResult processData(long dataUnitId, EmbeddedData embeddedData, boolean
produceReport)

CstsResult processData(long dataUnitId, EmbeddedData embeddedData, Time
earliestDataProcessingTime, Time latestDataProcessingTime, boolean produceReport)

CstsResult requestReset()
```

3.4.6 IInformationQuery

```
CstsResult queryInformation(ListOfParameters listOfParameters)
```

3.4.7 ICyclicReport Interface

```
CstsResult requestCyclicReport(long deliveryCycle, ListOfParameters listOfParameters)

CstsResult endCyclicReport()
```

3.4.8 INotification Interface

```
CstsResult requestNotification(ListOfParameters listOfEvents)

CstsResult endNotification()
```

3.4.9 IThrowEvent Interface

```
CstsResult requestExecution(ObjectIdentifier directiveIdentifier, DirectiveQualifier
directiveQualifier)

void actionCompletedSuccesfully(IExecuteDirective executeDirective)

void actionCompletedUnsuccesfully(IExecuteDirective executeDirective)

void acknowledgeDirective(IExecuteDirective executeDirective)

void declineDirective(IExecuteDirective executeDirective)
```

4 ARCHITECTURAL DESIGN

This section introduces architectural aspects of the CSTS API.

4.1 CSTS API Instances

An instance of the CSTS API is composed by multiple Service Instances. Service Instances consist of one Association Control Procedure, one prime procedure of any type and multiple secondary procedures. Figure 3: CSTS API Architecture illustrates the architecture of the CSTS API.

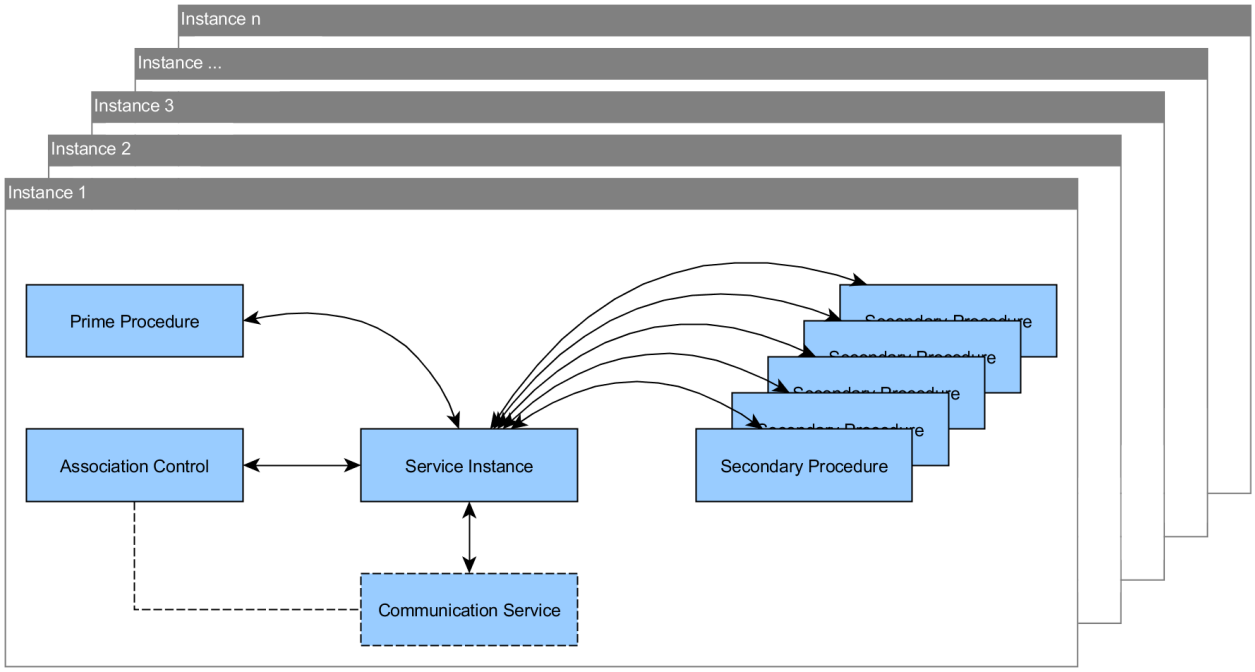

Figure 3: CSTS API Architecture

4.2 Procedures

Figure 4: Procedure Architecture illustrates the architecture of Procedures within the CSTS API. The Information Query and the Notification Procedures are chosen for demonstration purposes of how stateless (Information Query) and stateful (Notification) Procedures are integrated into the CSTS API architecture.

Note: continuous lines represent inheritance, dashed lines the implementation of an Interface.

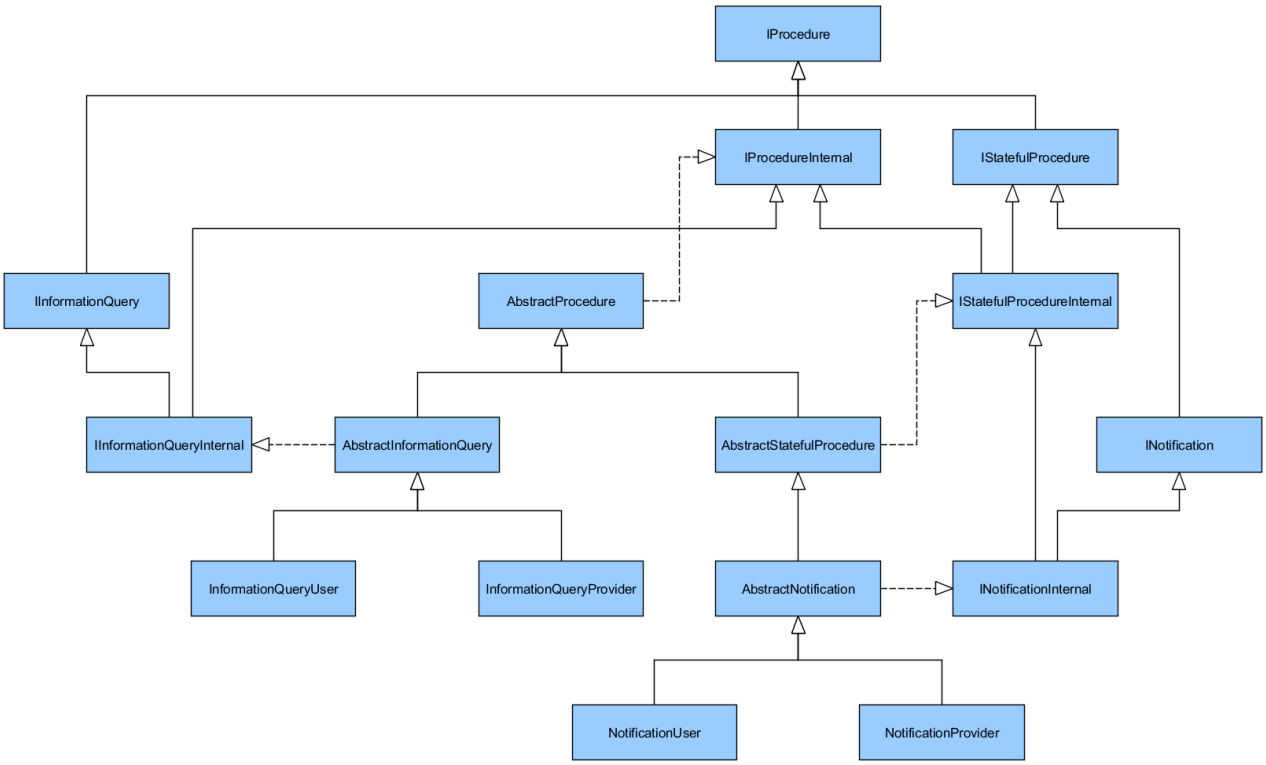

Figure 4: Procedure Architecture

The **AbstractProcedure** class is the base class of all procedures. It transitionally implements the **IProcedure** interface through the **IProcedureInternal** interface. All stateful procedures inherit from the **AbstractStatefulProcedure**, which transitionally implements the **IStatefulProcedure** interface through the **IStatefulProcedureInternal** interface.

All procedures (abstract as well as concrete) introduce their own interfaces for referring that procedure from outside the CSTS API (e.g. from the application) and additionally implement an internal interface. The internal interface specifies methods which are intended to be used solely from within the CSTS API, as in such cases as the underlying state machine.

5 WORKFLOWS WITHIN THE CSTS API

This section provides information about automated workflows within the CSTS API.

5.1 State Machine

Figure 5: State Machine shows the workflow of the underlying state machine of procedures.

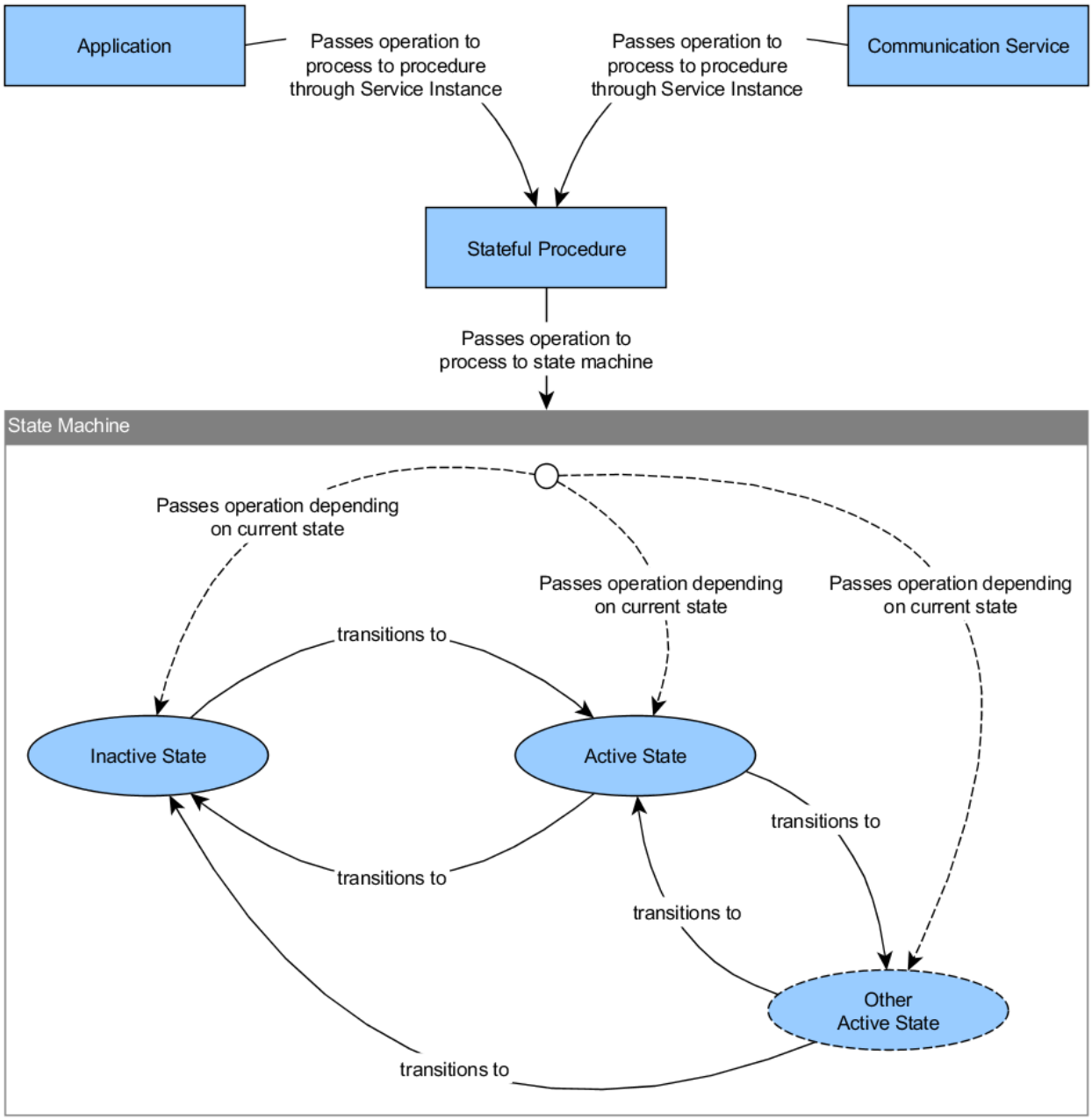

Figure 5: State Machine

Depending on the operation and its state, after processing the operation is being forwarded to the application or the underlying communication service.

5.2 Parameters

Parameters and their workflow within the CSTS API is shown in Figure 6: CSTS Parameters.

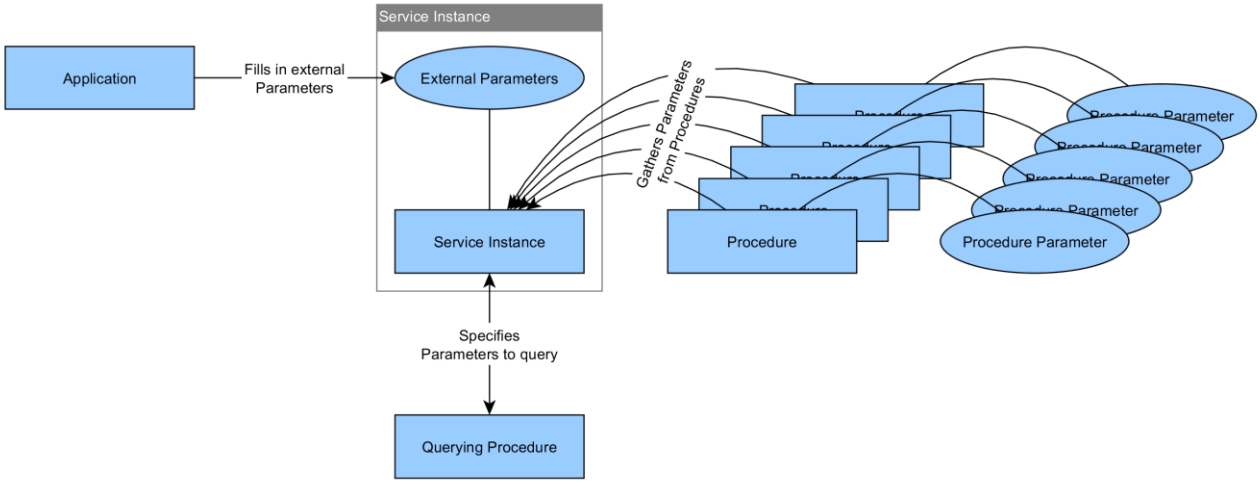

Figure 6: CSTS Parameters

Parameters belonging to procedures (configuration parameters) are being observed by the procedure by default. On change, an internal procedure event (not to be confused with CSTS events in Section 5.3) is triggered and the procedure gets a notification and can then further process the change (e.g. by raising a CSTS event).

External parameters are being passed to the service instance by the application. Passing parameters to the service instance is also possible at runtime. These parameters are immediately available to the querying procedures such as the **Information Query** procedure. The parameters are always queried through the service instance, where all parameters are gathered.

5.3 Events

CSTS events and their workflow is shown in Figure 7: CSTS Events. As parameters in Section 5.2, events follow a similar principle. Procedures can contain events and external events are being passed from the application to the service instance. Additionally the service instance contains events such as the production status change event.

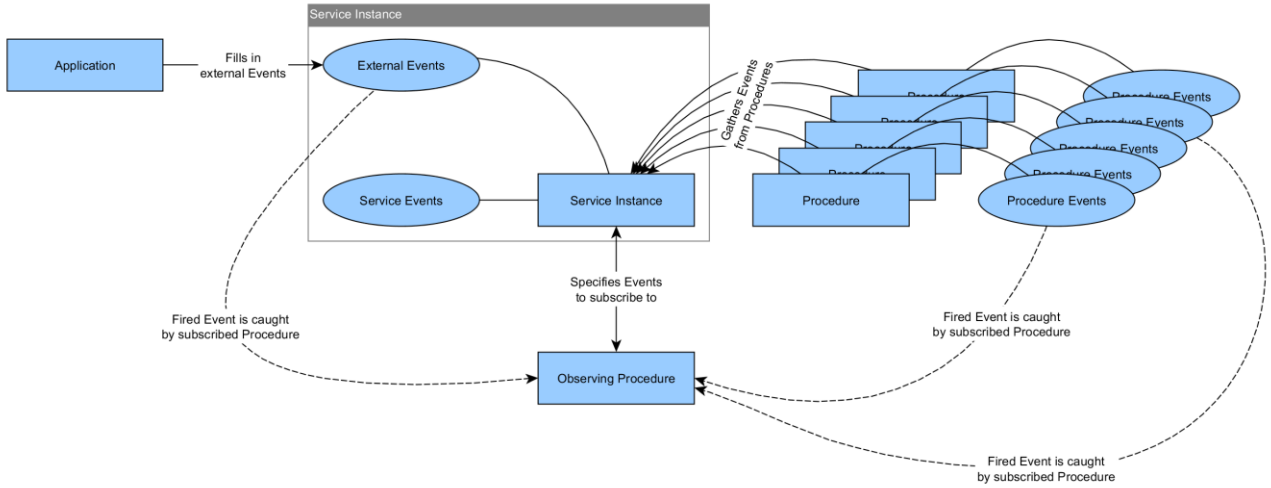

Figure 7: CSTS Events

Observing procedures can query all available events (or specific ones if desired) and put itself on the list of its observers. As soon as the event is fired, the observing procedure is notified and can use the received notification and its included event, which is composed by an identifier, a value and its time. These information can for example be used to create a NOTIFY operation and notify a user instance as in such procedures as the **Notification** procedure.

6 EXTENDING THE CSTS API

This section describes necessary steps to be taken when extending a procedure within the CSTS framework.

6.1 Suggested Approach

The inheritance hierarchy in Figure 8: Suggested Approach for extending Procedures is suggested to implement new procedures. The base procedure is the procedure being extended and the child procedure is the procedure being implemented.

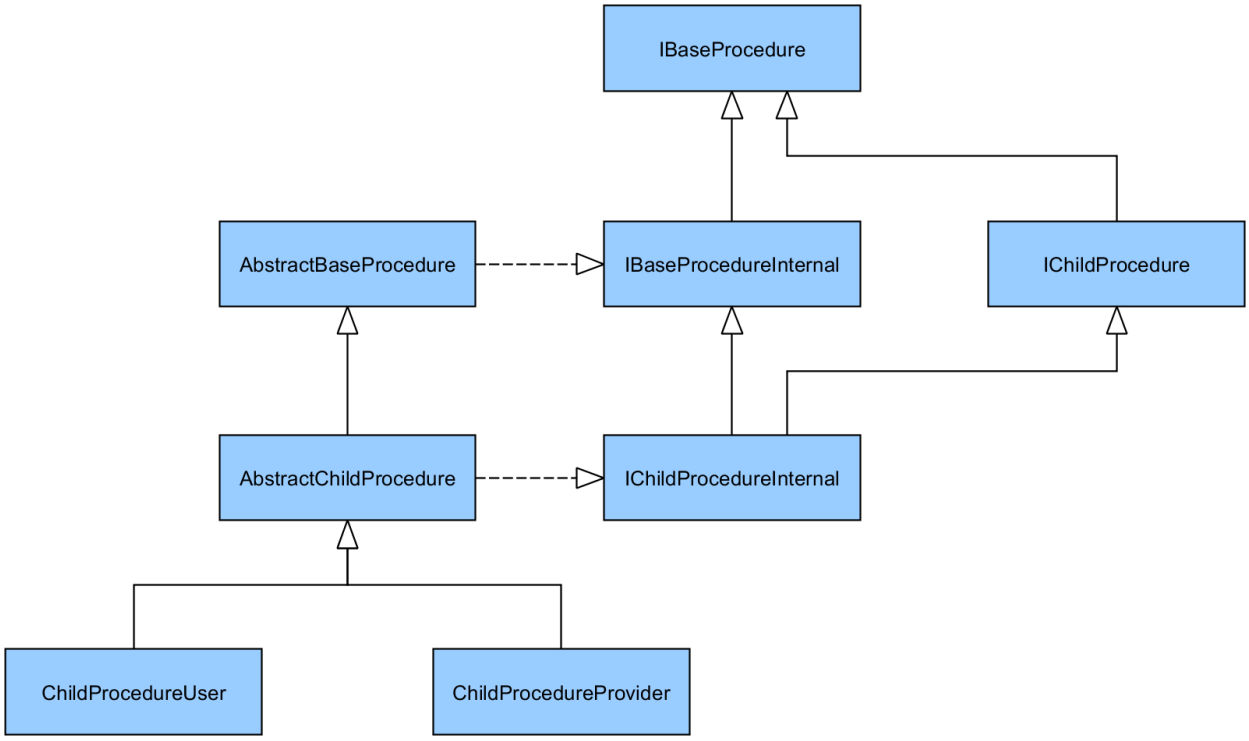

Figure 8: Suggested Approach for extending Procedures

Depending on the procedure, the CSTS API enforces all relevant methods to be implemented or already provides necessary methods which are optional to override. In case a procedure is open for extension at one point, the procedures already provide key methods to override for child procedures and integrate them in their workflow (e.g. the extension of the data processing status in the **Data Processing** procedure by the **Sequence-Controlled Data Processing** procedure).

6.2 Essential Methods

6.2.1 Initialization of Operation Types

Every procedure is forced to implement the method

```
protected abstract void initOperationTypes()
```

in which supported operation types are initialized. Supported operation types can be added by the

```
protected void addSupportedOperationType(OperationType type)
```

method. No more actions are needed to be taken by the procedure to support operation types.

6.2.2 Type and Version

The methods

```
public ProcedureType getType()
```

```
public int getVersion()
```

are enforced by the CSTS API to return the type and the version of the procedure. The **ProcedureType** class wraps an object identifier (see 7.1).

6.2.3 Termination of Procedures

All procedures are forced to implement the

```
public void terminate()
```

method which should cleanup procedure internals and initialize its state.

6.2.4 Creation of Operations

All procedures are able to create operations through their own factory methods. The operations will have valid internals and are ready to be parametrization and extension. The methods to create an operation are prefixed by *create*:

```
protected IBind createBind()
```

```
protected IUnbind createUnbind()
```

```
protected IPeerAbort createPeerAbort()
```

```
protected IStart createStart()
```

```
protected IStop createStop()

protected ITransferData createTransferData()

protected IProcessData createProcessData()

protected IConfirmedProcessData createConfirmedProcessData()

protected INotify createNotify()

protected IGet createGet()

protected IExecuteDirective createExecuteDirective()
```

6.2.5 *Initiation and Information of passed Operations*

Depending on the procedure and its role, following methods can be used to override and implement the functionality of how operations passed from the service instance or application are processed:

```
public CstsResult forwardInvocationToProxy(IOperation operation)

public CstsResult forwardAcknowledgementToProxy(IAcknowledgedOperation
acknowledgedOperation)

public CstsResult forwardReturnToProxy(IConfirmedOperation confirmedOperation)

public CstsResult forwardInvocationToApplication(IOperation operation)

public CstsResult forwardAcknowledgementToApplication(IAcknowledgedOperation
acknowledgedOperation)

public CstsResult forwardReturnToApplication(IConfirmedOperation confirmedOperation)
```

Provider procedures usually pass the operation to the underlying state machine (see 6.3), while user procedures simply forward operations to the service instance or application. The result of type **CstsResult** is an enumerated type to indicate the processing result of operations.

6.2.6 *Encoding and Decoding Operations*

The encoding and decoding with the use of generated Java classes read from the specified ASN.1 file, takes place in the following functions:

```
byte[] encodeOperation(IOperation operation, boolean isInvocation)

IOperation decodeOperation(byte[] encodedPdu)
```

These two methods are implemented by default in base classes and in many cases and an override should not be needed except in cases where additional functionality (*note: extensions are not considered additional functionality*) needs to be added.

All procedures being inherited from provide protected methods with the prefix encode to override from inheriting classes and extend their internals.

6.3 Diagnostics

All operations when acknowledged or returned with a negative result need to have valid diagnostics set (also by the application in cases the implemented service is responsible for acknowledging or returning operations as in such cases as the Throw Event procedure).

This is done by the

```
void setDiagnostic(Diagnostic diagnostic)
```

```
void setDiagnostic(EmbeddedData diagnosticExtension)
```

methods. The **EmbeddedData** class is used for extensions on any occasions within the CSTS API (see 7.9).

Many operations (e.g. the START operation) have their own diagnostics, where provided methods automatically encodes their diagnostics into the higher-level diagnostics. If procedures implement their own diagnostics, they need to make sure, they are encoded and passed to the corresponding diagnostic structure within the operation to ensure correct encoding.

6.4 Procedure States

All stateful procedures are automatically suitable for usage in the context of state machines. All procedures are forced to initialize its state. The abstract and parametrized class **State** represents all states and serves as the base class for all states. The generic parameter passed to the base class is an interface inheriting from **IStatefulProcedureInternal**.

All states are forced to implement the method

```
public CstsResult process(IOperation operation)
```

where the processing of passed operations take place and a result is returned.

The constructor of a state takes the procedure itself as an argument to guarantee access to the procedure and its internals within the state machine. Querying the current state and transitioning to another state can be done by the following methods of stateful procedures:

```
State<? extends IStatefulProcedureInternal> getState()
```

```
void setState(State<? extends IStatefulProcedureInternal> state)
```

Since states in the CSTS framework are not applicable in the role of a user, the API provides the class **UserState**, to initialize user roles with a dummy state. The user procedures are required to pass the operations to the underlying communication service or the application by bypassing the state machine.

The forwarding of operations take place in the following six methods:

```
CstsResult forwardInvocationToProxy(IOperation operation)
```

```
CstsResult forwardReturnToProxy(IConfirmedOperation confirmedOperation)
```

```
CstsResult forwardAcknowledgementToProxy(IAcknowledgedOperation acknowledgedOperation)
```

```
CstsResult forwardInvocationToApplication(IOperation operation)
```

```
CstsResult forwardReturnToApplication(IConfirmedOperation confirmedOperation)
```

```
CstsResult forwardAcknowledgementToApplication(IAcknowledgedOperation
acknowledgedOperation)
```

These methods are available to all procedures, stateful as well as stateless, user as well as provider and from within as well as from outside the underlying state machine.

6.5 Parameters

The CSTS API provides structures and mechanisms to implement parameters which are automatically integrated and used within the framework.

Figure 9: Architecture of Parameters shows the structure of parameters and procedure configuration parameters.

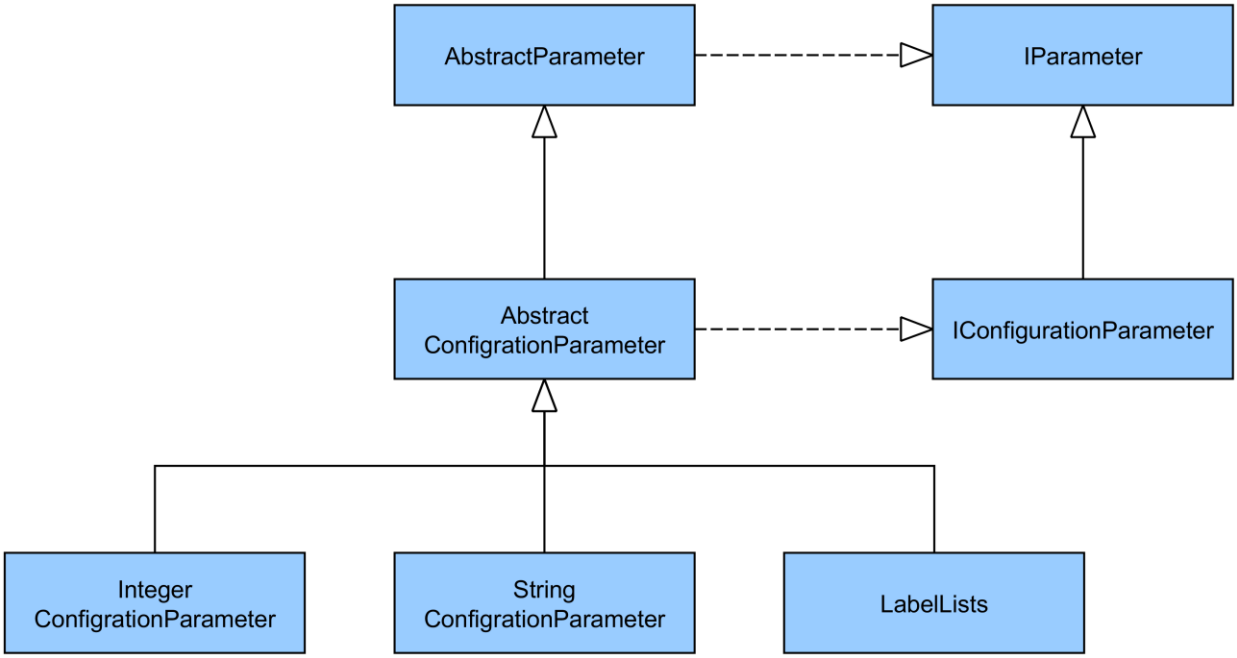

Figure 9: Architecture of Parameters

It is suggested to inherit from the **AbstractParameter** class if a new parameter type is being added or to inherit from the **AbstractConfigurationParameter** class in case new procedure configuration parameters are added to the CSTS framework.

Since parameters can be observed and procedure configuration parameters are being observed by their belonging procedures by default, these classes and their interfaces conform the structure and workflows within the CSTS API.

6.6 Procedure Configuration Parameters

Every procedure is able to implement configuration parameters. Procedure configuration parameters conform the **IConfigurationParameter** interface. The

```
protected void initializeConfigurationParameters()
```

method should be overridden by child procedures if configuration parameters need to be initially added to the procedure. Therefore the method

```
protected void addConfigurationParameter(IConfigurationParameter configurationParameter)
```

is used to add the configuration parameter to the procedure. Also the configuration parameter is being added to an internal list where it is being observed by the procedure. In case the configuration parameter has been changed the

```
protected void processConfigurationChange(IConfigurationParameter parameter)
```

is being called automatically, where the overriding procedure and process on the changed configuration parameter.

6.7 Procedure Events

Procedures are able to add procedure events and subscribe to other events and being notified in case they are being fired.

To initially add procedure events the method

```
protected void initializeEvents()
```

needs to be overridden. Events can be added to and removed from the procedure by the

```
protected void addEvent(ObjectIdentifier event)
```

```
protected void removeEvent(ObjectIdentifier event)
```

```
protected void removeEvent(IEvent event)
```

methods. Procedures can subscribe to or unsubscribe from events (independent if procedure events, or service events, or any other events) by calling the

```
protected void observeEvent(IEvent event)
```

```
protected void unsubscribeFromEvent(IEvent event)
```

methods. If a subscribed event is being fired, the

```
protected void processIncomingEvent(IEvent event)
```

gets called automatically and the procedure overriding that method is able to process on that event.

6.8 New Standards

The major change between version 1 and version 2 of the CSTS API is that the new release provides support for B1 and B2 version of the CSTS standard. The system has been designed and refactored in such a way that allows the introduction of new versions without major changed of the existing codebase and its structure.

To add support for further standards it is necessary to follow the following steps:

1. Place the .ASN1 grammar file in the resource folder and update the pom file to ensure the generation of the sources
2. Add the new version in the enumerate SfwVersion.java
3. Implement all the operations required by the new standard, and update the OpsFactory class such that objects with the new implementation are created when needed.
4. Implement a new codec for each procedure that is supported (codecs are located in the same package of the procedures)

7 PREDEFINED CCSDS TYPE CLASSES

This section introduces Java equivalents of predefined CCSDS types and how to use them in applications as well as in extended procedures.

7.1 Object Identifier

The CSTS API provides the **ObjectIdentifier** class as a representation of all object identifiers used and to be used within the API and extending procedures as well as applications built upon the API.

There are two possibilities of creating object identifier. By stating all numbers sequentially as in

```
ObjectIdentifier css = ObjectIdentifier.of(1, 3, 112, 4, 4)
```

or by using previously defined object identifier as a prefix as in

```
ObjectIdentifier csts = ObjectIdentifier.of(css, 1)
```

where the prefixed identifier is being extended by the numbers provided after.

The **OIDs** class provides all defined object identifiers of the provided ASN.1 file of the CSTS framework as static object identifier. It can be referenced as in the following:

```
ProcedureType type = ProcedureType.of(OIDs.associationControl)
```

The CSTS API provides the **OIDReader** utility class which is able to generate the **OIDs** class from a passed ASN.1 file. The generation is possible as hierarchical as well as a flat object identifiers. The following sample demonstrates the difference:

```
public static final ObjectIdentifier csts = ObjectIdentifier.of(css, 1);
```

```
public static final ObjectIdentifier csts = ObjectIdentifier.of(1, 3, 112, 4, 4, 1);
```

7.2 Name

Names are instantiated by passing an object identifier and either a functional resource name or a procedure instance identifier. Therefore the two methods

```
public static Name of(ObjectIdentifier objectIdentifier, FunctionalResourceName
functionalResourceName)
```

```
public static Name of(ObjectIdentifier objectIdentifier, ProcedureInstanceIdentifier
procedureInstanceIdentifier)
```

are provided by the CSTS API.

7.3 Label

Labels (similar to Names) are instantiated by passing an object identifier and either a functional resource type or a procedure type. Again the two methods

```
public static Label of(ObjectIdentifier objectIdentifier, FunctionalResourceType
functionalResourceType)
```

```
public static Label of(ObjectIdentifier objectIdentifier, ProcedureType procedureType)
```

are provided by the CSTS API.

7.4 Label List

A Label list is used within the label lists procedure configuration parameter. It consists of a name of the type String, an indicator if the list is the default list and the list of labels.

The instantiation of a label list takes place in its constructor:

```
public LabelList(String name, boolean defaultList)
```

7.5 Qualified Parameters

Qualified Parameters consisting of the type and value type have direct equivalents in Java to aid the use and encoding of type and values. Figure 10: Qualified Parameters Mapping visualizes how the Java equivalents are mapped to the ASN.1 types.

Note: continuous lines represent a singular association, wide lines a multiple association and dashed lines an equivalent of the Java types in ASN.1.

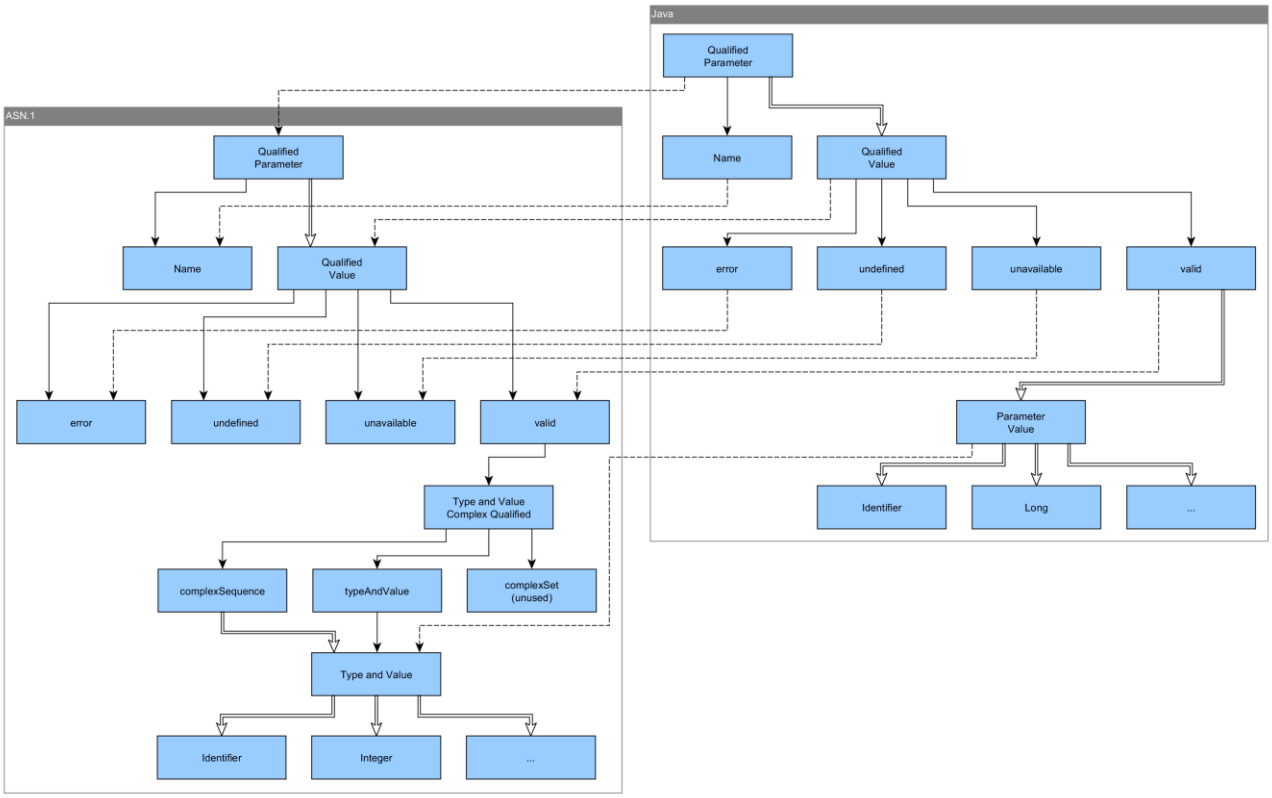

Figure 10: Qualified Parameters Mapping

The type of qualified values is also permissible to use within other structures such as the event value.

Qualified Parameters are specified by a name:

```
public QualifiedParameter(Name name)
```

Qualified values can be added to the underlying list of qualified values by specifying their qualifier, which is provided by an enumerated value:

```
public QualifiedValues(ParameterQualifier qualifier)
```

Parameter values are specified by their type:

```
public ParameterValue(ParameterType type)
```

Depending on the type, a list of a suitable type is instantiated which is shown in Figure 11: Parameter Type Mapping.

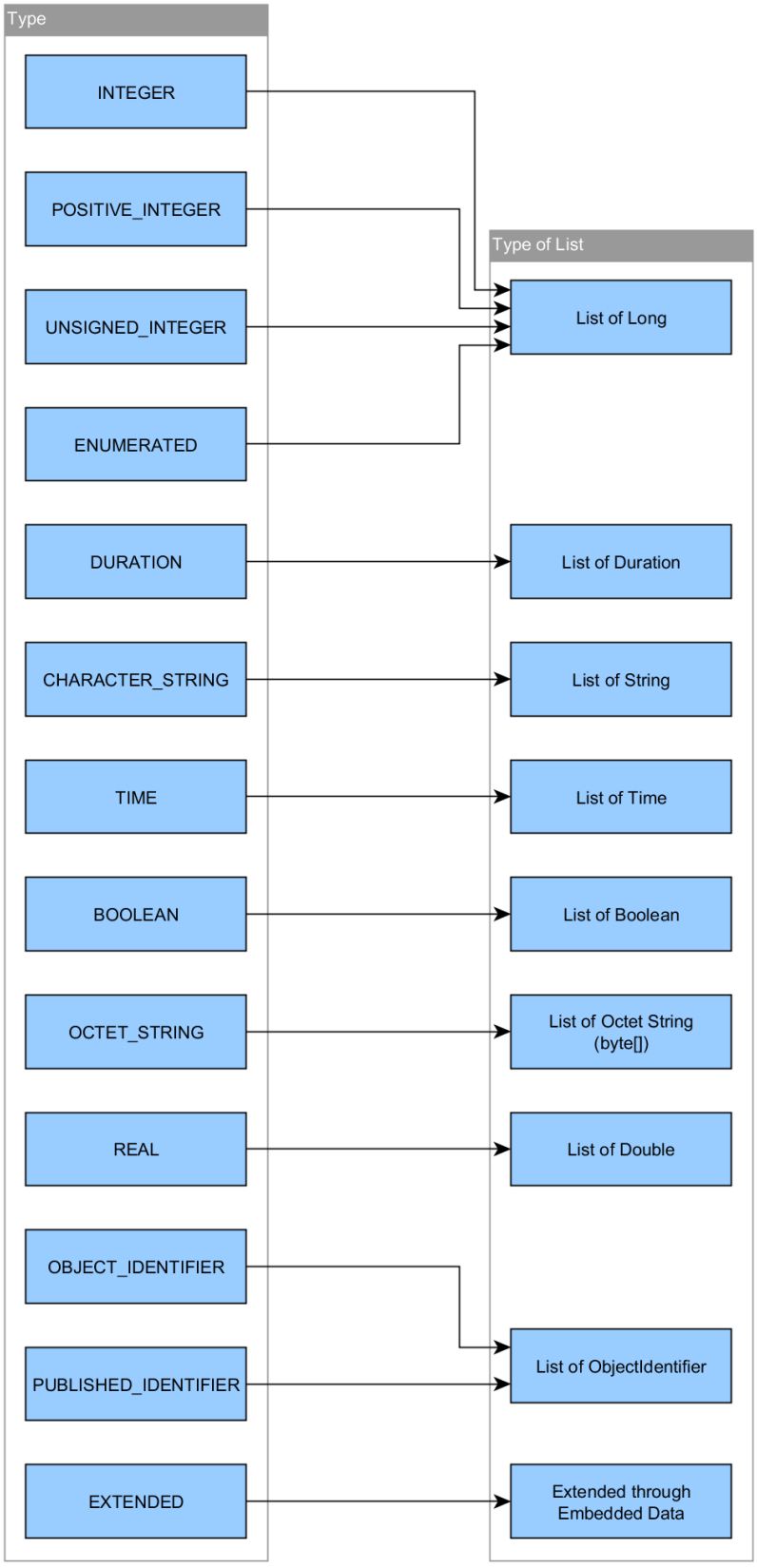

Figure 11: Parameter Type Mapping

7.6 Time

The **Time** class wraps the underlying byte array of length 8 or 10, depending on the choice of millisecond or picosecond resolution.

The **Time** class provided by the CSTS API is entirely compatible to the Java 8 Date/Time API.

Times can be instantiated by the following provided factory methods:

```
public static Time of(byte[] value)
```

```
public static Time of(LocalDateTime localDateTime)
```

```
public static Time of(Instant instant)
```

Furthermore the two methods

```
public static Time now()
```

```
public static Time nowLocal()
```

provide two possibilities to create a Time object of the current time, either in UTC or the local date and time from the system-clock.

The **Time** class implements the Comparable interface and is therefore comparable by default.

7.7 Conditional Time

The conditional time class wraps a time object as an optional value. Conditional times can be created using the following provided factory methods:

```
public static ConditionalTime unknown()
```

```
public static ConditionalTime of(Time time)
```

7.8 Duration

The **Duration** class wraps the duration of the Java 8 Date/Time API and provides its functionality in a CSTS context. A duration object is instantiated by specifying its type and a Java 8 duration object:

```
public static Duration of(DurationType type, java.time.Duration duration)
```

Note that Java 8 duration object does not have a specified SI unit and provide an interchangeable usability, depending on how they are accessed. The specified enumerated type is only significant to the CSTS API during encoding/decoding.

The **Duration** class implements the Comparable interface and is therefore comparable by default.

7.9 Extension

The **Extension** class is a key type being used in all aspects of the CSTS API where extension is necessary. Extensions wrap the underlying structure of the type **EmbeddedData** as an optional value. Embedded data types are composed by an object identifier and a byte array.

Extension can be created using the following factory methods:

```
public static Extension notUsed()
```

```
public static Extension of(EmbeddedData embeddedData)
```

An embedded data can also be created by its own factory method:

```
public static EmbeddedData of(ObjectIdentifier oid, byte[] data)
```