Implementing a Case Management Modeling and Notation (CMMN) System using a Content Management Interoperability Services (CMIS) compliant repository

Mike A. Marin\textsuperscript{1,2}, and Jay A. Brown\textsuperscript{1}
\textsuperscript{1} IBM Analytics Group
3565 Harbor Blvd., Costa Mesa, CA 92626 U.S.A.
{mikemarin,jay.brown}@us.ibm.com
\textsuperscript{2} University of South Africa
Pretoria, South Africa

April 28, 2015

Abstract

This paper describes how a Case Management Modeling and Notation (CMMN) implementation can use Content Management Interoperability Services (CMIS) to implement the CMMN information model. The interaction between CMMN and CMIS is described in detail, and two implementation alternatives are presented. An integration alternative where any external CMIS repository is used. This alternative is useful to process technology vendors looking to integrate with CMIS compliant repositories. An embedded alternative where a CMIS repository is embedded within the CMMN engine. This alternative is useful to content management vendors implementing CMMN. In both alternatives a CMIS folder is used as the case file containing the case instance data. The CMIS repository can also be used to store the CMMN models to take advantage of CMIS versioning and meta-data. Extensive Java pseudocode is provided as an example of how a CMMN implementation can use a CMIS repository to implement the CMMN information model. No extensions to CMIS are needed, and only minor extensions to CMMN are proposed.

Keywords: Case Handling, Case Management, Case Management System, Case Management Modeling and Notation, CMMN, CMMN Implementation, Content Management, Content Management System, Content Management Interoperability Services, CMIS

1 Introduction

In May 2014, the Object Management Group (OMG) formally released version 1.0 of the Case Management Modeling and Notation (CMMN) \cite{10} standard specification. The specification is intended to support case management applications \cite{7}. CMMN is based on two models, a behavioral model and an informational model. The CMMN specification indicates that the information model can be implemented using the Content Management Interoperability Services (CMIS) \cite{9} specification, however no details are given. This paper addresses that gap by describing how an CMMN implementation can use CMIS effectively. This paper is intended for implementors of CMMN, and should be read in conjunction with the CMMN specification \cite{10} and the CMIS specification \cite{9}. Familiarity with the CMMN and CMIS specifications is assumed.

Case management \cite{4,6,11} is intended to support the needs of knowledge workers when engaged in knowledge intensive goal oriented processes. It is common for knowledge workers to interact via documents (e.g. text documents, word processor documents, spreadsheets, presentations, correspondence, memos, videos,
| CMMN information model class | Corresponding CMIS class |
|------------------------------|--------------------------|
| CaseFile                    | cmis:folder              |
| CaseFileItem                | cmis:object              |
| CaseFileItemDefinition      | cmis:object Type         |
| Property                    | cmis:property Type       |

Table 1: Mapping CMMN information model to CMIS meta-model

pictures, etc.). Case management shares most of the knowledge intensive processes characteristics as defined by Di Ciccio et. al. which are knowledge driven, collaboration oriented, unpredictable, emergent, goal oriented, event driven, constraint and rule driven, and non repeatable [5]. Therefore, it makes sense that a platform to support knowledge workers provide content management and collaboration capabilities. Case management is defined by Forrester as:

"A highly structured, but also collaborative, dynamic, and information-intensive process that is driven by outside events and requires incremental and progressive responses from the business domain handling the case. Examples of case folders include a patient record, a lawsuit, an insurance claim, or a contract, and the case folder would include all the documents, data, collaboration artifacts, policies, rules, analytics, and other information needed to process and manage the case." [4]

This paper starts with a short introduction to CMMN in section 2 and CMIS in section 3. These introductions describe the main concepts, classes, and objects that will be used in the rest of the paper. Section 4 describes the two implementation alternatives. Section 5 describes how the CMMN information model could be implemented in a CMIS repository. Section 6 describes the implications for the CMMN models and for process interchange of case models. An example is given in Section 7. The example describes some of the functionality the end users will observe in a CMMN implementation that uses a CMIS repository as described in this paper. Conclusions are presented in section 8. Two appendixes are included. Appendix A shows the CMMN and CMIS meta-models for reference purposes. Finally, appendix B provides an example Java pseudocode showing a possible implementation of the CMMN information model in CMIS.

2 Case Management Modeling and Notation (CMMN)

The CMMN information model starts with a CaseFile that contains CaseFileItems. The important classes in the information model (see Figure 10) are,

**CaseFile:** The container for all the information in a case instance. The information in a CaseFile can be structured like discrete properties and variables, or unstructured like documents, pictures, voice recordings, video clips, etc. There is a single CaseFile in a case instance. It seems natural to implement the CaseFile as a folder (or directory) in a content management system (or file system).

**CaseFileItem:** A piece of information in a case instance. All the CaseFileItems of a case instance are stored in the case’s CaseFile. A case instance may have a large number of CaseFileItems. When using content management system (or file system), it seems natural to implement each CaseFileItem as a document or a folder. In most content management systems, both folders and documents have properties that can be used to store structured information. For example, a folder could be used to represent a customer. That folder may have properties like the name, customer number, phone number, physical and email address of the customer, etc. The folder may be used to store all the emails and documents related with that customer. That folder and all its information maybe part of a case instance, and so stored in the CaseFile.

**CaseFileItemDefinition:** Corresponds to the type of a CaseFileItem.
Property: Corresponds to a property or field of a CaseFileItem. A CaseFileItem may have many properties. In a content management system, properties are often referred as the meta-data of the documents or folders in the system.

The CMMN information model is shown in Figure 9 and Figure 10. Figure 9 shows the high level case model, and Figure 10 shows the details of the case file model. Note that a CaseFileItem has two self-referencing relationships:

- A composition relationship between parent and children that can be used to represent a folder structure, where the folder (CaseFileItem) contains either documents (other CaseFileItems) or other folders (also CaseFileItems).
- A reflexive association between sourceRef and targetRef that can be used to represent relationships between documents or folders.

2.1 Implementation using CMIS

In CMIS, we will use a folder to represent the case instance’s CaseFile, and it will contain all the case CaseFileItems for that case instance. The CaseFileItems will be documents or other folders. In CMIS as in CMMN, CaseFileItems (documents or folders) are typed. A CaseFileItem is an instance of a CaseFileItemDefinition. This model can be easily mapped into CMIS as described in Table 1.

Although there are multiple alternatives to implement the CMMN information model using CMIS, this paper explores just two alternatives:

- The integration alternative where an external CMIS repository is used. This alternative will be attractive to process technology vendors that want their technology to integrate with one or more existing CMIS compliant repositories.
- The Embedded alternative where a CMIS repository is embedded within the CMMN engine. This alternative will be attractive to content management vendors implementing CMMN over their CMIS compliant repository.
In both cases, the CMIS repository is used to store all or part of the CMMN information model. In both cases, the design tool could create CMIS declarations (mutable types), and the runtime user interface may provide access to both the CMMN engine and the CMIS repository. Figures 1, and 2 show a high level view of the two options.

An example of a simple case instance is shown in Figure 3. There are five entities in the figure, two \texttt{cmis:folders} and three \texttt{cmis:documents}. Each entity starts with three text lines. The first line indicates the name of the entity. The second line indicates the CMIS object that implements the entity, and the third line indicates the CMMN object that is being implemented. For illustration purposes, each entity has two or three properties. The example shows a case file instance (\texttt{CaseFile}) with four \texttt{CaseFileItem}s. Data A is a \texttt{cmis:document} that is being used for structured data so it has no document content (blob). From a CMIS perspective, Data A is a \texttt{cmis:document} that is missing the \texttt{ContentStream} (See the CMIS meta-model in Figure 11). This type of documents are normally called contentless documents, and they could be implemented as \texttt{cmis:item} instead of \texttt{cmis:document}. Incoming documents is a folder used to store picture B and document C. Both picture B and document C are real documents with blobs. Picture B is an image of a house, and document C is a report. There is a relationship between document C and picture B, as probably picture B is mentioned in the report.

For simplicity purposes this paper assumes the full CMMN information model is stored in a CMIS repository. Some implementations may decide to implement part of the information model in another database. The Java pseudocode presented in this paper is intended as an example on how a CMMN implementation may access CMIS, and it is not intended to be used as is. It is assumed that \texttt{CaseFile} can contain properties as any other \texttt{cmis:folder}, which is not allowed in CMMN. Hints on how to implement security, versions, and the mapping of CMIS events to CMMN events are given but not fully described in the Java pseudocode.

3 Content Management Interoperability Services (CMIS)

The CMIS specification is an open standard for dealing with Enterprise Content Management (ECM) repositories. It defines a common domain model and a set of three protocol bindings, each exposing the same domain model but serving a different type of client (SOAP, AtomPub, and JSON). OASIS approved CMIS as an official Specification in May 2010 and CMIS 1.1 was approved in December 2012 [9] and it remains the current version of the specification. The specification states:

"The CMIS interface is designed to be layered on top of existing Content Management systems and their existing programmatic interfaces. It is not intended to prescribe how specific features should be implemented within those CM systems, nor to exhaustively expose all of the CM
system’s capabilities through the CMIS interfaces. Rather, it is intended to define a generic/universal set of capabilities provided by a CM system and a set of services for working with those capabilities.” [9]

The CMIS domain model consists of a set of nine services (below) and a data model (also discussed below). It does not attempt to address all aspects of a typical ECM repository (e.g. Administrative functions, workflow, etc.) rather just the functions typically used at runtime by ECM client applications. The model’s nine services are detailed below.

### 3.1 CMIS Services and Data Model

CMIS defines a set of 9 services which include:

**Repository Services:** Used to discover info and capabilities of the connected repository.

**Navigation Services:** Used to traverse the repository’s folder hierarchy.

**Object Services:** Used to perform Create Read Update and Delete functions on objects.

**Multi-Filing Services:** Allows for multi-filing documents (not folders) within folders.
Discovery Services: Exposes Query (based on SQL 92) and getChanges which returns accumulated changes to the repository for indexers.

Versioning Services: Used to checkout documents and work with document versions.

Relationship Services: Used to discover and manage an object’s relationships.

Policy Services: Used to apply, remove, and query for policies.

ACL Services: Used to discover and manipulate ACLs (and ACEs) on an object.

The CMIS data model consists of a Repository object that reports all of the capabilities of the ECM repository, and a hierarchy of objects that are stored in the repository (see Figure 11). Each of these objects has a corresponding type or property definition (Shown in Figure 12). The important objects of the data model are,

- **cmis:object**: Base for cmis:document, cmis:folder, cmis:relationship, cmis:policy, and cmis:item.
- **cmis:document**: Describes documents in a content management system. Documents can be of any type, including pictures, video, voice recordings, work processor documents, spreadsheets, etc.
- **cmis:folder**: Describes folders in a content management system. CMIS supports a hierarchical folder structure.
- **cmis:relationship**: Can be used to establish a relationship between any two objects.
- **cmis:policy**: Is an object that can be applied to other cmis:objects. The CMIS specification does not describe any specific behavior for these objects.
- **cmis:item**: Used to model other object types that does not fit document, folder, relationship or policy types. An example could be content less documents.
- **cmis:secondary**: Define a set of properties that can be dynamically applied to an object. They can be used as markers to create dynamic collections of objects.

CMIS implementations exist for the most popular platforms and languages in Apache Chemistry [1]. These include libraries for Java, .Net, Python, JavaScript, PHP, Android and iOS (Objective-C). Finally a complete client development guide [8] and a server development guide [3] are available in addition to all the materials available on the Apache Chemistry site [1].

4 Alternatives

In this section the two alternatives are described, integration and embedded alternatives. The integration alternative will be appealing to process technology vendors, because it allows their CMMN implementations to use external CMIS repositories from other vendors. This may allow those process technology vendors to support one or more CMIS compliant repositories. The embedded alternative will be appealing to content management vendors, because it allows them to implement CMMN within their CMIS compliant repositories. Content management vendors entering the process space may prefer the embedded alternative.

4.1 Integration alternative

In this alternative the CMIS repository is external to the CMMN engine, and it could be implemented using any CMIS 1.1 compliant repository. This may allow a CMMN implementation to be compatible with one or more CMIS 1.1 repositories. Figure 1 shows a high level view of the integration alternative showing the touch points required to implement it. The external CMIS repository is used to store all or part of the CMMN information model.
Figure 4: Extending CMMN meta-model for CMIS Integration (integration alternative)

Figure 5: Integrated CaseFile
Figure 6: Extending CMMN meta-model for embedding CMIS (embedded alternative)

Figure 4 shows the CMMN meta-model extended to integrate with a CMIS repository, and Figure 5 shows a case file implemented by a \texttt{cmis:folder}. To support the integration alternative, the CMMN meta-model needs to be enhanced by adding references to the CMIS objects, as follows:

- The \texttt{CaseFile} now has a \texttt{CMISObjectId} attribute to reference the \texttt{cmis:folder}'s \texttt{id} that implements the case instance in the CMIS repository.

- The \texttt{CaseFileItem} now has a \texttt{CMISObjectId} attribute to reference the corresponding \texttt{cmis:object}'s \texttt{id} in the CMIS repository. In addition, it also has a \texttt{index} to implement the multiplicity concept in CMMN.

- The \texttt{CaseFileItemDefinition} now has a \texttt{CMISTypeId} attribute to indicate the \texttt{id} of a \texttt{cmis:object} type.

- The \texttt{Property} now has a \texttt{CMISPropertyId} attribute to indicate the \texttt{id} of a \texttt{cmis:property} type.

The CMMN design tool and the CMMN runtime client user interface may or may not be aware of the CMIS repository. An implementation in which the CMMN design tool is aware of the CMIS repository may allow users to create CMIS declarations (mutable types) corresponding to different document or folder types (\texttt{CaseFileItem} types' \texttt{CaseFileItemDefinitions}). Because both \texttt{cmis:folders} and \texttt{cmis:documents} can contain properties, the complete CMMN information model can be implemented in CMIS. For example an implementation may use properties in the \texttt{cmis:folder} to store case \texttt{CaseFile} properties.

The runtime user interface could include a CMIS client, giving the end users the ability to inspect and modify all the cases (\texttt{CaseFile}'s \texttt{cmis:folders}) based on his or her level of security access. When an end user modifies a case (\texttt{cmis:folder}) by adding documents, folders, or modifying the case folder documents or folders or their properties. The corresponding events are raised by CMIS and the CMMN implementation should react by evaluating and triggering the correct sentries' onParts.
4.2 Embedded alternative

The *embedded* alternative is the implementation of both CMIS and CMMN in the same engine. In this alternative, the CMIS repository is embedded within the CMMN engine. Figure 2 shows a CMIS repository embedded in a CMMN implementation.

Figure 6 describes the merged meta-model between CMMN and CMIS. Figure 7 shows the case file implemented by a *cmis:folder*. To support the *embedded* alternative, the CMMN meta-model needs to be enhanced as follows,

- The **CaseFile** becomes a generalization of *cmis:folder*.
- The **CaseFileItem** becomes a generalization of a *cmis:object*, and now has a *index* to implement the multiplicity concept in CMMN.
- The **CaseFileItemDefinition** becomes a generalization of *cmis:Object Type*.
- The **Property** becomes a generalization of *cmis:property Type*.

Everything that can be implemented with the *integration* alternative is also possible in *embedded* alternative. In addition, the *embedded* alternative provides advantages over the *integration* alternative. In particular the CMMN and CMIS design and runtime information may be stored in the same database. For example, event propagation can be done more efficiently, because a push model could be implemented versus the pull model described in this paper and in the Java pseudocode section B.3.

5 A CMIS repository as the CMMN information model

Independent of the alternative implemented (*integration* or *embedded*) the interaction between CMMN and CMIS will follow similar patterns and the CMMN implementation will use similar CMIS APIs. The simple way to use the CMIS API is to use one of the Apache Chemistry libraries [1]. Appendix B Java pseudocode shows Java pseudocode using the OpenCMIS Java API from Apache Chemistry [2] to describe at a high level how a CMMN implementation can invoke CMIS functionality.

It is important to notice that the CMMN meta-model in Figures 9 and 10 describe a modeling time meta-model that can be used for process interchange. While the CMIS meta-model in Figure 11 is a runtime model.
representing objects in a content management repository. The CMIS types meta-model in Figure 12 can be considered both a modeling and runtime meta-model. In here we combine all of those two meta-models and use them for both modeling and runtime execution.

5.1 CMMN Information Model

The information model in CMMN is based on a CaseFile (see Figure 9 and Figure 10). When implementing using CMIS, there are at least two options that can be used to implement the CaseFile.

Use a cmis:folder to represent the CaseFile. An integration alternative, as shown on Figure 5, may include a CMISObjectId in the CaseFile as a reference to the cmis:folder representing the case instance in CMIS. An embedded alternative, as shown in Figure 7, may implement the CaseFile as a generalization of a cmis:folder. Each case instance will have a cmis:folder containing all the case file items for that case. A cmis:folder can contain properties, and so, case properties can also be implemented in the cmis:folder implementing the CaseFile. The cmis:folder will probably outlive the case instance lifecycle, which is a good side effect, because all the case file items for a case instance will remain in the cmis:folder after the case is completed.

Use a database to implement the CaseFile, with references to its content in the CMIS repository. Under this option, there is no CMIS representation of the case file, and so, the implementation will need to keep track of the CMIS objects stored in the case (most likely documents and folders) by storing their cmis:objectId in the database.

This paper describes the first option of using a cmis:folder to represent the CaseFile. The CMMN information model matches well to the CMIS model. In both meta-models, CMMN and CMIS, there is a class that represents an object, a class that represents the type of that object, and a property class. Therefore, we can map between the two specifications as shown in Table 1. All the content in a CMIS repository can be represented by cmis:object(s) and their descendants. Similarly in CMMN the information model is represented by CaseFileItems. Therefore, we can map CaseFileItems to cmis:objects. That allows the CaseFileItems to describe all the CMIS objects, including documents and folders. Note that in CMMN, CaseFileItems representing folders use the children relationship (see Figure 10) to point to the CaseFileItems stored in the folder. As we described before, a CaseFile can be mapped to a cmis:folder. A CaseFileItemDefinition naturally maps into cmis:object Type, because CaseFileItemDefinition describes the type of a CaseFileItem, and so, it is similar to a cmis:object Type which defines the type of a cmis:object. Property and cmis:property Type represent the same concept. Therefore, the mapping in Table 1 allow us to describe the full CMMN information model using CMIS.

The only high level CMIS objects not included in Table 1 are cmis:policy, cmis:item, and cmis:secondary. They are optional in CMIS and probably not required for most CMMN implementations. However, below we describe how they could be implemented if needed, by indirectly mapping them to CaseFileItems (see Table 2).

We use cmis:folders to implement the CaseFileItem self-referencing composition relationship between parent and children (see Figure 10). We use cmis:relationship to implement the CaseFileItem self-referencing reflexive association between sourceRef and targetRef (see Figure 10).

5.1.1 Objects and data types

In CMMN a CaseFileItem can represent many objects, and the CaseFileItemDefinition defines the type by using a URI. In CMIS each object has its own class that is a specialization of cmis:object. Therefore to represent a CMIS object in CMMN, we need to set the correct URI value in the CMMN’s CaseFileItemDefinition’s definitionType, while assigning the correct CMIS object via cmis:object specialization to the caseFileItem.
| CMIS Object-type | CMMN Definition Type | CMMN CaseFileItemDefinition Type’s URI |
|------------------|----------------------|----------------------------------------|
| cmis:folder      | CMIS Folder          | http://www.omg.org/spec/CMMN/DefinitionType/CMISFolder |
| cmis:document    | CMIS Document        | http://www.omg.org/spec/CMMN/DefinitionType/CMISDocument |
| cmis:relationship | CMIS Relationship   | http://www.omg.org/spec/CMMN/DefinitionType/CMISRelationship |
| -- | XML-Schema Element | http://www.omg.org/spec/CMMN/DefinitionType/XSDElement |
| -- | XML Schema Complex Type | http://www.omg.org/spec/CMMN/DefinitionType/XSDComplexType |
| -- | XML Schema Simple Type | http://www.omg.org/spec/CMMN/DefinitionType/XSDSimpleType |
| -- | Unknown              | http://www.omg.org/spec/CMMN/DefinitionType/Unknown |
| -- | Unspecified          | http://www.omg.org/spec/CMMN/DefinitionType/Unspecified |

Extend CMMN as follows

| cmis:policy | -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISPolicy |
| cmis:item   | -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISItem |
| cmis:secondary | -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISSecondary |

other CMIS Object-types

| cmis:object | any of: |
|-------------|---------|
| CMIS Folder | http://www.omg.org/spec/CMMN/DefinitionType/CMISFolder |
| CMIS Document | http://www.omg.org/spec/CMMN/DefinitionType/CMISDocument |
| CMIS Relationship | http://www.omg.org/spec/CMMN/DefinitionType/CMISRelationship |
| -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISPolicy |
| -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISItem |
| -- | http://www.omg.org/spec/CMMN/DefinitionType/CMISSecondary |

Table 2: Object Types
| Type      | CMIS Type   | CMMN Property Type’s URI                  |
|-----------|-------------|------------------------------------------|
| string    | xsd:string  | http://www.omg.org/spec/CMMN/PropertyType/string |
| boolean   | xsd:boolean | http://www.omg.org/spec/CMMN/PropertyType/boolean |
| integer   | xsd:integer | http://www.omg.org/spec/CMMN/PropertyType/integer |
| float     | --          | http://www.omg.org/spec/CMMN/PropertyType/float |
| double    | --          | http://www.omg.org/spec/CMMN/PropertyType/double |
| duration  | --          | http://www.omg.org/spec/CMMN/PropertyType/duration |
| dateTime  | xsd:dateTime | http://www.omg.org/spec/CMMN/PropertyType/dateTime |
| time      | --          | http://www.omg.org/spec/CMMN/PropertyType/time |
| date      | --          | http://www.omg.org/spec/CMMN/PropertyType/date |
| gYearMonth| --          | http://www.omg.org/spec/CMMN/PropertyType/gYearMonth |
| gYear     | --          | http://www.omg.org/spec/CMMN/PropertyType/gYear |
| gMonthDay | --          | http://www.omg.org/spec/CMMN/PropertyType/gMonthDay |
| gDay      | --          | http://www.omg.org/spec/CMMN/PropertyType/gDay |
| gMonth    | --          | http://www.omg.org/spec/CMMN/PropertyType/gMonth |
| hexBinary | --          | http://www.omg.org/spec/CMMN/PropertyType/hexBinary |
| base64Binary | --       | http://www.omg.org/spec/CMMN/PropertyType/base64Binary |
| anyURI    | xsd:anyURI  | http://www.omg.org/spec/CMMN/PropertyType/anyURI |
| QName     | --          | http://www.omg.org/spec/CMMN/PropertyType/QName |

Extend CMMN as follows

| -- | xsd:decimal | http://www.omg.org/spec/CMMN/PropertyType/decimal |
| -- | Id          | http://www.omg.org/spec/CMMN/PropertyType/Id |
| -- | HTML        | http://www.omg.org/spec/CMMN/PropertyType/HTML |

Table 3: Property types

Objects
Table 2 compares the object types in CMIS with the CMMN information model. The object types in CMMN are defined by the DefinitionTypeEnum. The CMMN’s CaseFileItemDefinition describes the type of the CaseFileItem using an URI that includes some CMIS types. Table 2 does an explicit mapping between CMIS object types and CMMN’s CaseFileItemDefinition types. The CMIS types described by the CMMN’s URIs should be enough for most implementations, but if needed three more URIs can be added for cmis:policy, cmis:item, and cmis:secondary, as described in Table 2. Note that CMIS policy, item, and secondary are optional in CMIS, and some implementions may not implement them.

Data types
CMMN and CMIS property types are based on the XML Schema types [12]. CMMN uses most of the XML Schema types, while CMIS uses a limited set of types. This makes it easy to map CMIS types to CMMN. Table 3 maps the CMIS types onto CMMN types. To fully support CMIS, the CMMN property type URI needs to be extended with xsd:decimal, Id, and HTML.

5.1.2 Navigating the information model
CMMN describes a standard set of seven CaseFileItem operations (see the CMMN specification [10] section 7.3.1 CaseFileItem operations) for the behavioral model to navigate the information model. The Java pseudocode in section B.1 CaseFile navigation operations in appendix B Java pseudocode shows a potential implementation of these operations. All the operations described here work over CaseFileItems in a case instance. As described before, all the CaseFileItems are contained within the CaseFile of the case instance (see Figure 10). All the operations return ether a CaseFileItem (see Table 2) instance; or an Element which corresponds to a property of a CMIS object (see Table 3). Note that to implement CMMN’s multiplicity
for CaseFileItems an index has been added to CaseFileItem for both integration or embedded alternatives (see Figure 5, Figure 7, Figure 6, and Figure 4). The index must be maintained by the implementation and should be incremented when multiple cmis:objects within the same case instance (CaseFile) have the same cmis:name. For most implementations that may imply that a CMIS property index must be added to all the cmis:objects that can be stored in a case folder (CaseFile).

The operations defined in the CMMN specification are intended to be used in CMMN expressions. Therefore these operations are intended to be implemented as part of the CMMN expression support (see the CMMN specification [10] section 5.4.7 Expressions). The default expression language in CMMN is XPath, however CMMN implementations may support other expression languages. Note that an implementation will need to wrap the operations shown in section B.1 to expose them in the supported expression languages. Therefore, the Java pseudocode in appendix B Java pseudocode is intended as an example, and may not implement the CMMN operations exactly as they will be exposed in an expression language.

5.1.3 Modifying the information model

The previous section describes how to implement the required CMMN operations to navigate a case instance (CaseFile) information model using the CMIS API. This section will describe how to use the CMIS API to modify a case instance (CaseFile) information model. The Java pseudocode in section B.2 CaseFile modification operations in appendix B Java pseudocode shows an implementation of operations to create CaseFileItems and relationships between them. Operations to create documents (createCaseFileItemDocumentInstance), folders (createCaseFileItemFolderInstance), and relationships (createCaseFileItemRelationship) are described.

The createCaseFileItemFolderInstance is used to create cmis:folders to implement the CaseFileItem self-referencing composition relationship between parent and children. The createCaseFileItemRelationship is used to create cmis:relationships to implement the CaseFileItem self-referencing reflexive association between sourceRef and targetRef. Those two CaseFileItem self-referencing relationships are shown in Figure 10.

The update and deletion of cmis:documents, cmis:folders, or cmis:relationships can be trivially accomplished using Apache Chemistry OpenCMIS [2], with existing method calls and so, are not included in the Java pseudocode.
Table 4: CMIS folder events

| CMIS folder event | CMMN CaseFileItem event | Description |
|-------------------|-------------------------|-------------|
| file in folder    | addChild                | A new object has been added to the folder |
| create relationship | addReference           | A new cmis:relationship to the folder has been added |
| create folder     | create                  | The folder has been created |
| delete folder     | delete                  | The folder has been deleted |
| unfile document   | removeChild             | An object has been removed (un-filed) from the folder |
| delete relationship | removeReference       | A cmis:relationship that pointed to the folder was removed |
| delete + create   | replace                 | The complete folder was replaced with a new version |
| update folder     | update                  | The folder properties have been modified |

Table 5: CMIS document, relationship, policy, item, or secondary events

| CMIS document, relationship, policy, item, or secondary event | CMMN CaseFileItem event | Description |
|---------------------------------------------------------------|-------------------------|-------------|
| create relationship                                           | addReference            | A new cmis:relationship to the object has been added |
| create                                                        | create                  | The object has been created |
| delete                                                        | delete                  | The object has been deleted |
| delete relationship                                           | removeReference         | A cmis:relationship that pointed to the object was removed |
| delete + create                                               | replace                 | A new version of the object has replaced the previous version |
| update                                                        | update                  | The object properties have been modified |

5.2 CaseFileItem Lifecycle event propagation

For CMMN’s sentries to work correctly, events generated from the CMIS objects must be propagated to the corresponding sentry onPart. CMMN describes a lifecycle for the CaseFileItem as shown in Figure 8. From a CMIS perspective, we can separate the lifecycle state transitions between cmis:folders as described in Table 4, and other CMIS objects (cmis:document, cmis:relationship, cmis:policy, cmis:item, and cmis:secondary) as described in Table 5.

The Java pseudocode in section B.3 Event propagation in appendix B Java pseudocode shows how to pull the CMIS repository for events. Those CMIS events then can be used to evaluate and trigger sentry’s onPart. Calling the GetContentChangesForEventPropagation method in section B.3 Event propagation, will place the thread into an infinite loop pulling for CMIS events. The implementor will need to complete the PushChangeEvents Java pseudocode method to propagate the events into the CMMN implementation. OpenCMIS Enum ChangeType has only four values for change events, CREATED, DELETED, SECURITY, and UPDATED. So the developer will have to map these to the the events in Table 4 and Table 5. This exercise is left to the reader.

5.3 Versioning and Roles

Although CMIS supports versioning, the CMMN specification states that for purposes of CMMN modeling, only the last version is assumed, but implementations can use versioning if required (see the CMMN specification [10] section 5.3.2.1 Versioning). When implementing CMMN using CMIS, it makes sense to take advantage of the CMIS versioning capabilities.

Roles in CMMN are used for human tasks and are not associated with CaseFileItems, however when using CMIS it makes sense to use the CMIS security features to support the CMMN role concept. Each CMIS object (cmis:object) can have a ACL associated with it to implement security.
6 CMMN models

This section describes how to store the CMMN models in the CMIS repository. It also describe the effects of using CMIS as described in this paper on process interchange.

6.1 Storing the CMMN Models

The CMIS repository can be used by the CMMN modeler tool to store the models. The modeler tool can take advantage of the versioning offered by most CMIS repositories to maintain the versions of its models. It can also take advantage of the CMIS folders to create project folders with the ability to create sub-folders to store the multiple assets of a project. In general, the CMIS repository can be used as the modeler repository for CMMN models and other modeling artifacts. The CMMN models and other artifacts can be represented as cmis:documents and stored in specialized cmis:document Types and cmis:folder Types. The CMMN model documents can have specialized meta-data for the CMMN modeler tool to use. For example, project name, department, etc. Standard CMIS meta-data can also be used by the CMMN modeler tool to keep track of its models. For example, cmis:name, cmis:description, cmis:createdBy, cmis:creationDate, cmis:lastModifiedBy, cmis:lastModificationDate, cmis:versionLabel, etc.

6.2 CMMN Extensions

In order for the CMMN implementation to take full advantage of the capabilities offered by CMIS, few extensions to CMMN are required, as follows.

Property types can be extended as shown in Table 3 to support xsd:decimal, Id, and HTML types. Note that if a CMMN application is exclusively using a CMIS repository then it would never encounter one of these types. So these extensions may be optional.

CaseFileItem types may need to be extended as shown in Table 2. This is optional, because not all implementations will need to support all the CMIS objec types. Implementations that need to support cmis:policy, cmis:item, or cmis:secondary will need to extend the CaseFileItemDefinition definitionType's URI as described in Table 2.

Extended attributes are needed in both alternatives. The embedded alternative requires extended attributes to support,

- index as an attribute of CaseFileItem

The integration alternative requires extended attributes to support,

- CMISObjectID as an attribute of CaseFile and CaseFileItem
- index as an attribute of CaseFileItem
- CMISTypeId as an attribute of CaseFileItemDefinition
- CMISPropertyId as an attribute of Property

In CMMN 1.0, these extensions affect process interchange. Future versions of the CMMN specification may introduce extensible attributes and rules on how to preserve extended URIs in CaseFileItemDefinition definitionType's URI and Property Type's URI.

Currently, tools wishing to preserve CMIS 1.0 process interchange may need to introduce an option when saving CMMN models to indicate if the model must be CMMN 1.0 compatible, and if so, the following transformations will be required, to remove extensions:

- Remove the extended attributes as follows,

  index from CaseFileItem
CMISObjectId from CaseFile and CaseFileItem
CMISTypeId from CaseFileItemDefinition
CMISPropertyId from Property

- Map extended Property Types as follows,

  xsd:decimal (http://www.omg.org/spec/CMMN/PropertyType/decimal) to double (http://www.omg.org/spec/CMMN/PropertyType/decimal)
  xsd:Id (http://www.omg.org/spec/CMMN/PropertyType/Id) to string (http://www.omg.org/spec/CMMN/PropertyType/Id)
  xsd:HTML (http://www.omg.org/spec/CMMN/PropertyType/HTML) to string (http://www.omg.org/spec/CMMN/PropertyType/HTML)

- Map extended CaseFileItemDefinition definitionTypes as follows,

  cmis:policy (http://www.omg.org/spec/CMMN/DefinitionType/CMISPolicy) to Unknown (http://www.omg.org/spec/CMMN/DefinitionType/Unknown)
  cmis:item (http://www.omg.org/spec/CMMN/DefinitionType/CMISItem) to Unknown (http://www.omg.org/spec/CMMN/DefinitionType/Unknown)
  cmis:secondary (http://www.omg.org/spec/CMMN/DefinitionType/CMISSecondary) to Unknown (http://www.omg.org/spec/CMMN/DefinitionType/Unknown)

- Review the generalizations from CMIS classes in the *embedded* alternative, which are,

  CaseFile generalization of cmis:folder
  CaseFileItem generalization of cmis:object
  CaseFileItemDefinition generalization of cmis:Object Type
  Property generalization of cmis:property Type

7 Example

This example describes an hypothetical CMMN implementation using a CMIS repository to implement the case file and to store CMMN models, as described in this paper. In this example, the implementation has two end user front end tools, the modeling tool and the client tool. Both front ends may be integrated into a single user interface. The modeling tool allows users to create CMMN case models, and so, implements the design time aspects of CMMN. The modeling tool is used by business analysts or case workers to create, update, and manage CMMN models. Case models are serialized into machine readable files as described in the CMMN specification. The files could be XMI or CMMN XML-Schema (XSD) compliant files. Those files are stored in the CMIS repository as documents. The client tool allows case workers to interact with a case instance, and so, implements the runtime aspects of a CMMN implementation. Case workers using the client tool are able to create case instances, interact with case instances by adding content, executing tasks and stages, engaging in planning by adding discretionary items to the case instance plan, collaborating with other case workers to complete case instances, etc. The case instance information model is implemented in CMIS as cmis:folder representing the case file. Therefore, each case instance will have its unique CMIS folder. The user using the client tool can see the state of the case instance in the CMIS folder and associated content. An example of a case file is shown in Figure 3. In that figure, the case instance for project XX has a CaseFileItem Data 1 with some properties, and a sub-folder for incoming documents with two documents, a house picture and a report document.

In a system with a clear separation between design and runtime, a business analyst may create a case model and save it in the CMIS repository using the modeling tool. The modeling tool may expose the CMIS
versioning capability. Taking advantage of these capabilities, the business analysts may maintain multiple versions of the case model and may decide to deploy to a production system one of those versions.

In a system with no separation between design and runtime, a case worker may create a CMMN model starting from scratch or using a template stored in the CMIS repository. In both cases, the resulting model may be stored in the CMIS repository for future usage as a template. In systems with no clear separation between design and runtime, models will normally start incomplete and will evolve as the case workers process the instance. These case models will continually evolve, and so, the version capabilities of CMIS will be used to keep track of the evolution of the model.

Eventually a case instance will be created and case workers will collaborate to complete the case using the client tool. Documents of multiple types maybe required to process the case instance. For example, emails, word processing documents, spreadsheets, pictures, videos, voice recordings, case comments, etc. Those documents will be stored in the case folder. To organize those documents, the case workers may decide to create a folder structure under the case folder. For example, it may be useful to create a sub-folder for correspondence. That correspondence sub-folder may be further subdivided into an incoming correspondence sub-folder and an outgoing correspondence sub-folder.

In addition to the client tool that allows the case workers to interact with the case instance, other CMIS client programs could also interact with the case folder. Documents in the case instance may be created by the case workers or it may be placed in the case instance by computer programs using the CMIS API to access the case file. Events are raised when documents are added to the case, are modified, or are removed. Because both documents and folders are CaseFileItem, those events can be used in entry or exit criterion to tasks, stages, or milestones. So, as the case file is modified by either the case workers using the client tool or CMIS clients interact with the case file, then entry or exit criterion may be triggered.

8 Conclusion

This paper described how to implement the CMMN information model using CMIS. There is no need to extend CMIS to be used by CMMN, and only minor extensions to CMMN are proposed in this paper. Two implementation alternatives were described. An integration alternative where an external CMIS repository is used and an embedded alternative where a CMIS repository is embedded within the CMMN engine. The integration alternative will be appealing to process technology vendors, and the embedded alternative will be appealing to content management vendors. In both cases, the CMIS repository can be used to store the CMMN models to take advantage of CMIS versioning and meta-data. Extensive sample Java pseudocode is provided and analysis of the meta-models was done to guide implementors.

References

[1] Apache Chemistry. Apache Chemistry. http://chemistry.apache.org/, 2015. Online; accessed 1-February-2015.

[2] Apache Chemistry. Apache Chemistry OpenCMIS. http://chemistry.apache.org/java/opencmis.html, 2015. Online; accessed 1-February-2015.

[3] J. Brown and F. Muller. OpenCMIS Server Development Guide: Building custom CMIS servers with the Apache Chemistry OpenCMIS Server Framework. https://github.com/cmisdocs/ServerDevelopmentGuidev2, 2nd edition edition, 2014. Online; accessed 18-December-2014.

[4] L. C. Clair, C. Moore, and R. Vitti. Dynamic Case Management - An Old Idea Catches New Fire. Technical report, Forrester, Cambridge, MA, 2009.

[5] C. Di Ciccio, A. Marrella, and A. Russo. Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. Journal on Data Semantics, 4(1):29–57, 2015.
A CMMN and CMIS Meta-models

The CMMN and the CMIS meta-models are provided here for reference purposes. The four figures shown here have been copied from the formal specifications [9, 10].

Figure 9 describes the CMMN high level meta-model showing the relationship between the Case and the CaseFile that implement the CMMN information model. Figure 10 describes how the CaseFile contains all the CaseFileItems in the case. In addition, it shows that CaseFileItems can be used to create a folder structure using the composition relationship between parent and children; and it also shows that relationships between CaseFileItems can be implemented using the reflexive association between sourceRef and targetRef. These CMMN meta-models describe a CMMN model at modeling time and can be used for process interchange.

Figure 11 describes the CMIS objects meta-model, and Figure 12 describes the CMIS type system. These CMIS meta-models describe a content repository runtime, by describing the objects stored in the content repository at execution time.
Figure 9: CMMN High level meta-model

Figure 10: CMMN case file item meta-model
Figure 11: CMIS meta-model
Figure 12: CMIS Types
B  Java pseudocode

All the sample Java pseudocode present here is uses Apache Chemistry OpenCMIS which is a standard
CMIS reference client library for Java [2]. This pseudocode is an example of how to use OpenCMIS to
implement the CMMN information model. It is not intended for production usage and so it lacks error
recovery pseudocode. There are few methods that use System.out.println in areas that are left as exercise
to the reader to complete the methods.

This appendix lists the complete Java pseudocode in file CaseFileItemOperations.java. It starts as follows,

```java
/*
 * Copyright 2015 Jay Brown & Mike Marin
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 * http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package discovery.common;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.apache.chemistry.opencmis.client.api.ChangeEvent;
import org.apache.chemistry.opencmis.client.api.ChangeEvents;
import org.apache.chemistry.opencmis.client.api.CmisObject;
import org.apache.chemistry.opencmis.client.api.ItemIterable;
import org.apache.chemistry.opencmis.client.api.ObjectId;
import org.apache.chemistry.opencmis.client.api.Property;
import org.apache.chemistry.opencmis.client.api.Relationship;
import org.apache.chemistry.opencmis.client.api.Session;
import org.apache.chemistry.opencmis.commons.PropertyIds;
import org.apache.chemistry.opencmis.commons.data.ContentStream;
import org.apache.chemistry.opencmis.commons.enums.BaseTypeId;
import org.apache.chemistry.opencmis.commons.enums.VersioningState;

B.1 CaseFile navigation operations

This section describes the CMMN standard set of CaseFileItem operations for the behavioral model to
navigate the information model (see the CMMN specification [10] section 7.3.1 CaseFileItem operations).

The class CaseFileItemOperations is used to define all the methods described in this paper. The class
constructor requires a CMIS session and a root folder that serves as the CaseFile for the case instance.
Most of the methods operate on a case instance (CaseFile). For illustration purposes, some methods in this
class can operate outside the case instance.

```
Assumes that on CMIS server the class type for
org.apache.chemistry.opencmis.commons.enums.BaseTypeId has had the cmnn:index
property added. If you are using a subclass of document to represent our
CaseFiles then substitute your class type name for BaseTypeId.CMIS_MESSAGE
in the code that follows.

All of this code is based on Apache Chemistry OpenCMIS objects. For the
JavaDocs for these objects please see
http://chemistry.apache.org/java/0.12.0/maven/apidocs/

A l l o f t h i s code i s based on Apache Chemistry OpenCMIS objects. For the
JavaDocs for these objects please see
http://chemistry.apache.org/java/0.12.0/maven/apidocs/

public class CaseFileItemOperations {

private Folder rootCaseFolder; // initialized to a folder object
private Session cmisSession; // initialized to a live session
public static final String indexPropertyName = "cmnn:index";

/**
 * @param rootCaseFolder
 * − the root folder holding all artifacts for a case
 * @param cmisSession
 * − live session to a CMIS server
 */
public CaseFileItemOperations(Folder rootCaseFolder, Session cmisSession) {
    this.rootCaseFolder = rootCaseFolder;
    this.cmisSession = cmisSession;
}

B.1.1 CaseFileItem instances

The CMMN specification describes two overloaded operations to navigate CaseFileItem instances. The
first has a single input parameter (itemName).

getCaseFileItemInstance(IN itemName : String,
OUT CaseFileItem instance)

Get a CaseFileItem (a cmis:object most likely a document or folder) instance with itemName (cmis:name)
within the CaseFile container. If no CaseFileItem instance for the given itemName exists, an empty
cmis:document (CaseFileItem) instance is returned. If more than one CaseFileItem instance name has
the same itemName (cmis:name), an arbitrary one should be returned.

This Java pseudocode provides three implementations for this operation. One returning a cmis:object
(getCaseFileItemInstance), one returning a cmis:document (getCaseFileItemDocumentInstance), and
finally one returning a cmis:folder (getCaseFileItemFolderInstance).

/**
 * General version that returns any type of object Searches for documents
 * first. (folders, documents, items)
 * This is implemented using Query without a join so it will be compatible
 * with repositories that do not support joins. This could also be
 * implemented using folder.getDescendants(<depth>)
 * @param itemName
 * @return
 */
public CmisObject getCaseFileItemInstance(String itemName) {
    return getCaseFileItemInstanceChild(this.rootCaseFolder, itemName);
}

23
```java
/**
 * Find the first descendant document that matches the name supplied or null
 * if none found.
 * @param itemName
 * @return
 */
public Document getCaseFileItemDocumentInstance(String itemName) {
    return getCaseFileItemDocumentInstanceChild(this.rootCaseFolder, itemName);
}

/**
 * Find the first descendant folder that matches the name supplied or null
 * if none found.
 * @param itemName
 * @return
 */
public Folder getCaseFileItemFolderInstance(String itemName) {
    return getCaseFileItemFolderInstanceChild(this.rootCaseFolder, itemName);
}

The second has two input parameters (itemName and index).

getCaseFileItemInstance(IN itemName : String,
            index : Integer,
            OUT CaseFileItem instance)

Get a CaseFileItem (a cmis:object most likely a document or folder) instance with itemName (cmis:name) and CaseFileItem’s index (see Figure 5 and Figure 7) within the CaseFile container. This operation is to be used for CaseFileItem (a cmis:object) instances with a multiplicity greater than one. The index is used to identify a concrete CaseFileItem (a cmis:object) most likely a document or folder) instance from the collection of CaseFileItem instances. If no CaseFileItem instance for the given itemName exists, or if the index is out of the range of CaseFileItem instances, an empty CaseFileItem instance is returned.

Note that Java does not provide methods overloading, so a number 2 was appended to the method names. This Java pseudocode provides three implementations for this operation. One returning a cmis:object (getCaseFileItemInstance2), one returning a cmis:document (getCaseFileItemDocumentInstance2), and finally one returning a cmis:folder (getCaseFileItemFolderInstance2).

```
/ **Note for this query to work you must have defined [indexPropertyName]**
// on your document and folder classes.
// If you have your index field defined on a subclass of document/folder
// change
String whereClause = "IN_TREE('" + this.rootCaseFolder.getId()
    + "') AND cmis:name='" + itemName + "' AND"
    + indexPropertyName + "= '" + index.toString() + "'";
ItemIterable<CmisObject> queryResult = this.cmisSession.queryObjects(
    BaseTypeId.CMIS_DOCUMENT.value(), whereClause, true, context);

// find the documents first
if (queryResult.getPageNumItems() == 0) {
    for (CmisObject tempObj : queryResult) {
        caseFileItem = tempObj;
        break;
    }
} else {
    // repeat for folder
    queryResult = this.cmisSession.queryObjects(BaseTypeId.CMIS_FOLDER.value(),
        whereClause, true, context);
    for (CmisObject tempObj : queryResult) {
        caseFileItem = tempObj;
        break;
    }
}
return caseFileItem;
}
public Folder getCaseFileItemFolderInstance(String itemName, Integer index) {
    Folder caseFileItem = null;
    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);

    // Note for this query to work you must have defined [indexPropertyName]
    // on your folder class.
    // If you have your index field defined on a subclass of document change
    // this query to select for that type.
    // instead of BaseType.id.CMIS_DOCUMENT
    String whereClause = '\"IN_TREE\"(" + this.rootCaseFolder.getId() + ";" + itemName + ";" + index + ";" + indexPropertyName + ";" + index.toDynamicObject() + ";" ;
    ItemIterable<CmisObject> queryResult = this.cmisSession.queryObjects(BaseTypeId.CMIS_FOLDER.value(), whereClause, true, context);

    for (CmisObject tempObj : queryResult) {
        caseFileItem = (Folder) tempObj;
        break;
    }
    return caseFileItem;
}

B.1.2 CaseFileItem properties

getCaseFileItemInstanceProperty (IN item : CaseFileItem instance,
                                    propertyName : String,
                                    OUT Element)

Get the value of a CaseFileItem instance property. If propertyName refers to a non-existing property of the CaseFileItem instance, an empty Element MUST be returned. The Element returned MUST be of the specified property type for the CaseFileItem instance.

/**
 * For retrieving a property from a fileInstance you only need to call the
 * .getProperty on the instance itself. As shown here in this method.
 */
public <T> Property<T> getCaseItemInstanceProperty(Document fileItemInstance,
                                           String propertyName) {
    return fileItemInstance.getProperty(propertyName);
}

/**
 * Returns a single value property object by name or the first of a list of
 * properties
 */
public fileItemInstance
* @param propertyName
* @return Native value for property, i.e. String, Integer, Boolean, etc.,
* ...
*/

```java
public Object getCaseItemInstancePropertySingleValue(Document fileItemInstance, String propertyName) {
    return fileItemInstance.getProperty(propertyName).getFirstValue();
}
```

**B.1.3 Using CaseFileItems as folders**

The methods in this section are used to navigate `cmis:folders` when they implement the `CaseFileItem` self-referencing composition relationship between `parent` and `children` (see Figure 10).

```java
getCaseFileItemInstanceChild(IN item : CaseFileItem instance,
                               childName : String,
                               OUT CaseFileItem)
```

Get a child `CaseFileItem` instance for a given `CaseFileItem` instance. This operation is valid for `CaseFileItems` implemented as `cmis:folders` (`cmis:folder`). The value of parameter `childName` specifies the name (`cmis:name`) of the child to get with in the `cmis:folder`. If no child of the given name exists for the `CaseFileItem` instance, an empty `CaseFileItem` instance is returned.

This operation is provided to navigate the composition relationship between `CaseFileItems` used to implement a folder structure. They are represented in the CMMN meta-model (see 10) by the `parent` and `children` composition relationship. This operation navigates from the `parent` (always a `cmis:folder`) to the `child` (most likely a `cmis:document` or folder).

This Java pseudocode provides three implementations for this operation. One returning a `cmis:object` (`getCaseFileItemInstanceChild`), one returning a `cmis:document` (`getCaseFileItemDocumentInstanceChild`), and finally one returning a `cmis:folder` (`getCaseFileItemFolderInstanceChild`).

```java
/**
 * For the folder instance passed in, return any child object matching the name
 * ...
 */

```
} else {
    // no documents found so let's do the same query for folders
    queryResult = this.cmisSession.queryObjects(BaseTypeId.CMIS_FOLDER.value(),
        whereClause, true, context);

    for (CmisObject tempObj : queryResult) {
        System.out.println("Object name: "+ tempObj.getName());
        caseFileItem = tempObj;
        break;
    }
}

return caseFileItem;

/**
 * For the folder instance passed in, return any child folder matching the
 * name (any depth)
 *
 * @param folderItemInstance
 * @param childName
 * @return
 */
public Folder getCaseFileItemFolderInstanceChild(Folder folderItemInstance, String childName) {
    Folder caseFileItem = null;
    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);

    String whereClause = "IN_TREE(" + folderItemInstance.getId() + ") AND cmis:name=" + childName + ");
    ItemIterable<CmisObject> queryResult = this.cmisSession.queryObjects(
        BaseTypeId.CMIS_FOLDER.value(), whereClause, true, context);

    if (queryResult.getPageNumItems() == 0) {
        for (CmisObject tempObj : queryResult) {
            System.out.println("Object name: "+ tempObj.getName());
            // Search was only for folders
            // so always ok to cast here
            caseFileItem = (Folder) tempObj;
            break;
        }
    }

    return caseFileItem;
}

/**
 * For the folder instance passed in, return any child document matching the
 * name (any depth)
 *
 * @param folderItemInstance
 * @param childName
 * @return
 */
public Document getCaseFileItemDocumentInstanceChild(Folder folderItemInstance, String childName) {
    Document caseFileItem = null;
    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);

    String whereClause = "IN_TREE(" + folderItemInstance.getId() + ") AND cmis:name=" + childName + ");
ItemIterable<CmisObject> queryResult = this.cmisSession.queryObjects(
    BaseTypeId.CMISDOCUMENT.value(), whereClause, true, context);

if (queryResult.getPageNumItems() == 0) {
    for (CmisObject tempObj : queryResult) {
        System.out.println("Object name:" + tempObj.getName());
        // Search was only for documents
        // so always ok to cast here
        caseFileItem = (Document) tempObj;
        break;
    }
}
return caseFileItem;

getCaseFileItemInstanceParent(IN item : CaseFileItem instance,
                                  OUT CaseFileItem instance)

Get the parent CaseFileItem (cmis:folder) instance of a CaseFileItem instance. Note in the worse case, the parent will be the CaseFile, which is the parent of all the CaseFileItems in a case.

This operation is provided to navigate the composition relationship between CaseFileItems used to implement a folder structure. They are represented in the CMMN meta-model (see 10) by the parent and children composition relationship. This operation navigates from the child (most likely a cmis:document or folder) to the parent (always a cmis:folder).

/**
 * Takes any CMIS object (document or folder) and returns first parent.
 * If this is a folder it can only have one parent. If it is a document it
 * may have multiple parents. (if multifiled)
 * @param caseFileItem
 * @return
 */
public Folder getCaseFileItemInstanceParent(FileableCmisObject caseFileItem) {
    return caseFileItem.getParents().get(0);
}

/**
 * Takes any CMIS object (document or folder) and returns list of parents.
 * If this is a folder it can only have one parent. If it is a document it
 * may have multiple parents. (if multifiled)
 * @param caseFileItem
 * @return
 */
public List<Folder> getCaseFileItemInstanceParents(FileableCmisObject caseFileItem) {
    return caseFileItem.getParents();
}

B.1.4 Relationships between CaseFileItems

The methods in this section are used to navigate the cmis:relationship used to implement the CaseFileItem self-referencing reflexive association between sourceRef and targetRef (see Figure 10).

getCaseFileItemInstanceSource(IN item : CaseFileItem instance,
                                  OUT CaseFileItem instance)

Get the source CaseFileItem instance of a CaseFileItem instance.
This operation is provided to navigate relationships between CaseFileItem instances. They are represented in the CMMN meta-model (see 10) by the sourceRef and targetRef relationship. This operation navigates from the targetRef to the sourceRef.

```java
/**
 * Return all cmis:relationship items associated with the supplied
 * CaseFileItem where the *source* is the object specified
 * @param caseFileItem
 */
public ItemIterable<CmisObject> getCaseFileItemInstanceSource(
    FileableCmisObject caseFileItem) {

    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);

    String whereClause = "cmis:sourceId = '" + caseFileItem.getId() + "'";

    // Query all relationships (even if relationship is for item outside of
    // this case)
    // this query requires that the repository have
    // 1) cmis:relationship objects
    // 2) cmis:relationship object with queryable=true
    ItemIterable<CmisObject> queryResult = this.cmisSession.queryObjects(
        BaseTypeId.CMIS_RELATIONSHIP.value(), whereClause, true, context);

    // any additional filtering or processing you want to do on the list of
    // relationships here...
    return queryResult;
}
```

getCaseFileItemInstanceTarget(IN item : CaseFileItem instance,
    targetName : String,
    OUT CaseFileItem instance)

Get a target CaseFileItem instance for a given CaseFileItem instance. The value of parameter childName specifies the name (cmis:name) of the target to get. If no target of the given name exists for the CaseFileItem instance, an empty CaseFileItem instance will be returned.

This operation is provided to navigate relationships between CaseFileItem instances. They are represented in the CMMN meta-model (see 10) by the sourceRef and targetRef relationship. This operation navigates from the sourceRef to the targetRef.

```java
/**
 * Return all cmis:relationship items associated with the supplied
 * CaseFileItem where the *target* is the object specified
 * @param caseFileItem
 */
public ItemIterable<CmisObject> getCaseFileItemInstanceTarget(
    FileableCmisObject caseFileItem) {

    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);

    String whereClause = "cmis:targetId = '" + caseFileItem.getId() + "'";

    // Query all relationships (even if relationship is for item outside of
    // this case)
    // this query requires that the repository have
    // 1) cmis:relationship objects
```
ItemIterable<CMisObject> queryResult = this.cmisSession.queryObjects(BaseTypeId.CMIS_RELATIONSHIP.value(), whereClause, true, context);

// any additional filtering or processing you want to do on the list of relationships here...
return queryResult;

/**
 * Return all cmis:relationship items associated with the supplied caseFileItem.
 * @param caseFileItem
 */
public ItemIterable<CMisObject> getCaseFileItemInstanceSourceOrTarget(FileableCMisObject caseFileItem) {
    OperationContext context = cmisSession.createOperationContext();
    context.setMaxItemsPerPage(100);
    context.setIncludeAllowableActions(Boolean.FALSE);
    String whereClause = "cmis:sourceId=" + caseFileItem.getId() + " OR (cmis:targetId=" + caseFileItem.getId() + ")";
    // Query all relationships (even if relationship is for item outside of this case)
    // this query requires that the repository have
    // 1) cmis:relationship objects
    // 2) cmis:relationship object with queryable=true
    ItemIterable<CMisObject> queryResult = this.cmisSession.queryObjects(BaseTypeId.CMIS_RELATIONSHIP.value(), whereClause, true, context);
    // any additional filtering or processing you want to do on the list of relationships here...
    return queryResult;
}

## B.2 CaseFile modification operations

This section shows some examples on how to use CMIS to modify the case instance (CaseFile) information model. Three creation methods are included here, two of them allow to create folders and documents in the root folder representing the case instance (CaseFile), and one to create relationships between CMIS objects. They can be used as examples of how the case information model can be modified.

Updates and deletions of objects in the case information model can be easily done using standard OpenCMIS [2] method calls in the corresponding objects.

/**
 * Create a new caseFileItemFolder instance
 * @param folderItemInstance
 * @param itemFolderName
 * @param objectType
 * @param additionalProperties
 * @return new caseFileItemFolder instance
 */
```java
public Folder createCaseFileItemFolderInstance(Folder folderItemInstance,
                                            String itemFolderName, String objectType,
                                            Map<String, Object> additionalProperties) {

    if (folderItemInstance == null) {
        folderItemInstance = this.rootCaseFolder;
    }

    if (additionalProperties == null) {
        // create a properties object
        additionalProperties = new HashMap<String, Object>();
    }

    // set the object type (specific subclass of cmis:relationship) – not required
    if (objectType != null) {
        additionalProperties.put(PropertyIds.OBJECT_TYPE_ID, objectType);
    } else {
        additionalProperties.put(PropertyIds.OBJECT_TYPE_ID,
                                  BaseTypeId.CMIS_FOLDER.value());
    }

    // set the name (required)
    additionalProperties.put(PropertyIds.NAME, itemFolderName);

    return folderItemInstance.createFolder(additionalProperties);
}

/**
 * Create a new caseFileItemDocument instance
 *
 * @param folderItemInstance
 *     – parent folder item for this new document (optional) If omitted document will be created as child of root caseItemInstance for this case.
 * @param itemFolderName
 *     – name of the new document
 * @param objectType
 *     – (optional) specific subtype of cmis:document to create If omitted cmis:document will be used
 * @param additionalProperties
 *     – (optional) any other optional properties to set on the object
 * @param versioningState
 *     – (optional) initial versioning state of the document.
 * @param contentStream
 *     – (optional) content Stream for the document. If omitted then document will contain only metadata (see additionalProperties)
 * @param additionalProperties
 *     – (optional) any other optional properties to set on the object
 *
 * @return new caseItemDocument instance
 */
public Document createCaseFileItemDocumentInstance(Folder folderItemInstance,
                                                    String itemDocumentName,
                                                    String objectType, VersioningState versioningState, ContentStream contentStream,
                                                    Map<String, Object> additionalProperties) {

    if (folderItemInstance == null) {
        folderItemInstance = this.rootCaseFolder;
    }

    if (additionalProperties == null) {
        // create a properties object
        additionalProperties = new HashMap<String, Object>();
    }
```
// set the object type (specific subclass of cmis:relationship) - not required
if (objectType != null) {
    additionalProperties.put(PropertyIds.OBJECT_TYPE_ID, objectType);
} else {
    additionalProperties.put(PropertyIds.OBJECT_TYPE_ID,
        BaseTypeId.CMIS_DOCUMENT.value());
}

// set the name (required)
additionalProperties.put(PropertyIds.NAME, itemDocumentName);

return folderItemInstance.createDocument(additionalProperties, contentStream,
    versioningState);

/**
 * Create a new relationship object and return it.
 *
 * @param source - source object to set on relationship
 * @param target - target object to set on relationship
 * @param objectType - optional - specific subclass of relationship required
 * @param additionalProperties - additional properties to set on the relationship - these
 *     must all be defined on the objectType you specify.
 *
 * @return The CMIS ObjectID of the new relationship object.
 */
public ObjectId createCaseFileItemRelationship(CmisObject source, CmisObject target,
    String objectType,
    Map<String, String> additionalProperties) {
    if (additionalProperties == null) {
        // create a properties object
        additionalProperties = new HashMap<String, String>();
    }

    // set the object type (specific subclass of cmis:relationship) - not required
    if (objectType != null) {
        additionalProperties.put(PropertyIds.OBJECT_TYPE_ID, objectType);
    } else {
        additionalProperties.put(PropertyIds.OBJECT_TYPE_ID,
            BaseTypeId.CMIS_RELATIONSHIP.value());
    }

    // set the source and target (required)
    additionalProperties.put(PropertyIds.SOURCE_ID, source.getId());
    additionalProperties.put(PropertyIds.TARGET_ID, target.getId());

    return cmisSession.createRelationship(additionalProperties);
}

/**
 * To update a folder or document use the object’s CmisObject.updateProperties(Map<String,?> properties) method directly.
 * To delete use the object’s .delete() method.
 *
 * See http://chemistry.apache.org/java/0.12.0/maven/apidocs/ for more details on the signatures of these methods.
 */
B.3 Event propagation

This section describes how to receive the events from the CMIS repository. The following methods are included in this class for illustration purposes, but these methods are not case instance specific. They will receive events from all the case instances in the CMIS repository. These methods should be executed in their own thread, because `GetContentChangesForEventPropagation` will go into an infinite loop. Most implementations will encapsulate the two methods shown in this section in another class to be executed in its own thread.

```java
/**
 * This following section is code to demonstrate event push
 * techniques for CMIS.
 * This code would be put into its own module where it could run
 * continuously
 * 
 * @param session − active OpenCMIS session with CMIS server
 * @throws InterruptedException due to use of Thread.sleep
 */
public void GetContentChangesForEventPropagation(Session session) throws InterruptedException {
    Integer BatchTimeIntervalInMS = 1000; // create a batch of changes every 1 second
    long maxChangesToProcessInSingleBatch = 500; // if you need to limit // batch size
    // if your repository supports property details with content changes
    // then you may set this to true (may affect performance)
    Boolean includePropertiesWithChangeEvents = false;
    // determine starting point (token) for changes
    String latestChangeToken = session.getRepositoryInfo().getLatestChangeLogToken();
    String previousChangeToken = "";
    System.out.println("Using initial change token:" + latestChangeToken);
    // main event processing loop − infinite
    while (true) {
        Thread.sleep(BatchTimeIntervalInMS);
        ChangeEvents changeEvents = session.getContentChanges(latestChangeToken,
            includePropertiesWithChangeEvents, maxChangesToProcessInSingleBatch);
        // push these changes to subscribed clients if there are new events
        // to process
        if (changeEvents.getChangeEvents().size() > 0) {
            // The changelog token can be the same and there still might
            // be one event in the list. This is so that clients can
            // validate that there is overlap and thus that they have not
            // missed an event. (verify there is no gap)
            if (!previousChangeToken.equals(latestChangeToken)) {
                PushChangeEvents(changeEvents);
            }
        }
    }
}
```
else {
    System.out.println("Duplicate overlap that we don’t have to process found.");
}
}
else {
    System.out.println("No changes found.");
}

// update our change token based on the last change we processed
previousChangeToken = latestChangeToken;
latestChangeToken = changeEvents.getLatestChangeLogToken();
System.out.println("updated changeToken: " + latestChangeToken);
}

/**
 * @param changeEvents
 * - openCMIS ChangeEvents holder
 * (org.apache.chemistry.opencmis.client.api.ChangeEvents)
 */
public void PushChangeEvents(ChangeEvents changeEvents) {
    // ... impl dependent code to push these events here
    for (ChangeEvent ce : changeEvents.getChangeEvents()) {
        // print statements just for debugging
        System.out.println("ID: " + ce.getObjectId());
        System.out.println("type: " + ce.getChangeType());
        // process each change event separately or in a batch
        // using whatever push technology your system requires. (e.g. Comet)
    }
} // End class CaseFileItemOperations