Java Application Programming Interface (API) for Annotation Imaging Markup (AIM)

Hakan Bulu[1], Daniel L. Rubin[2]

[1]Dokuz Eylul University, Department of Computer Engineering, Izmir, Turkey

[2] Stanford University, Department of Radiology and Center for Biomedical Informatics Research, Stanford, U.S.

1. Introduction

About AIM

Information Sciences in Imaging at Stanford (ISIS) Laboratory of Stanford University is currently funded by the National Cancer Institute of the National Institutes of Health to develop techniques and tools to enable extracting quantitative and semantic information from radiology images in a standard format called Annotation and Image Markup (AIM).

The goal of the AIM project is to develop a mechanism, for modeling, capturing, and serializing image annotation and markup data that can be adopted as a standard by the medical imaging community. The AIM project produces both human and machine-readable artifacts. One of the most important points about the AIM is storing the annotations as XML (Extensible Markup Language) format called as “AIM XML”. Additional applications transform the AIM XML into DICOM-SR, HL7-CDA XML and AIM ontology instances. The AIM model is shown in Figure 1. For more information about the AIM project; [1], [2], [3].

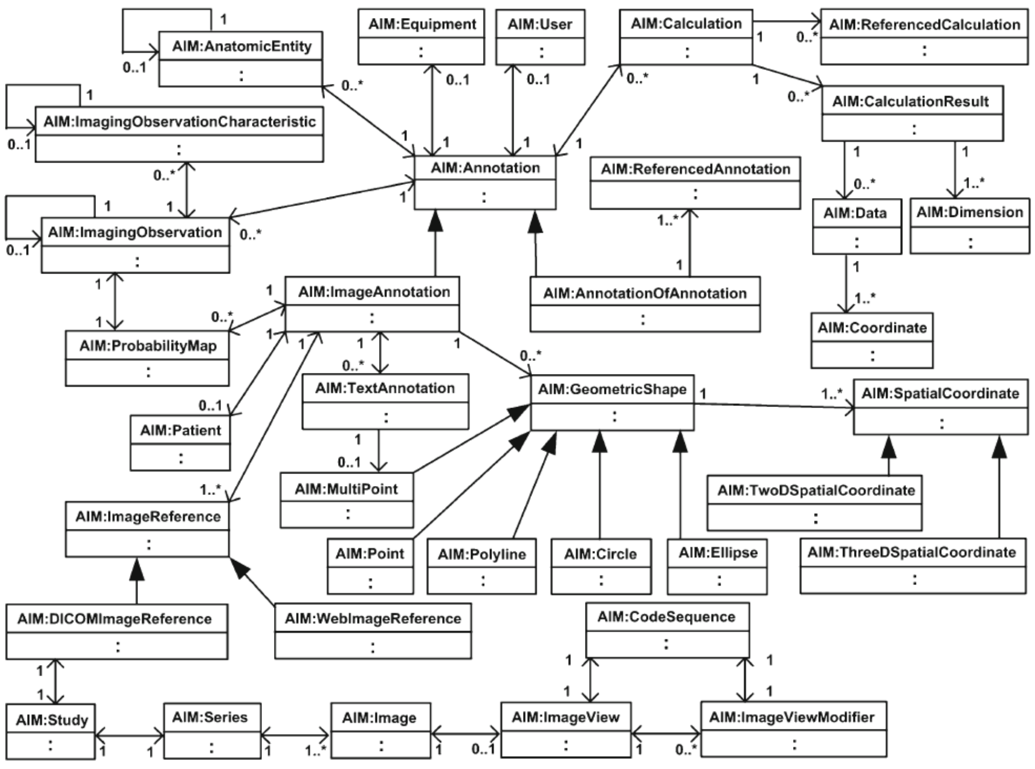

Figure 1 - AIM UML Class Diagram

Structure of the AIM is not small, so understanding this structure can be difficult when you become familiar. Also, if any developer/researcher wants to use AIM, he/she has to know how to parse and create XML files. In other words they have to know XML technologies. Because of that, to simplify the usage of AIM, we have developed an Application Programming Interface (API) called as “AIM API”. The AIM API [4] was developed in Java programming language and it creates an object model, using Java classes for each class in the AIM Schema. The class hierarchy of the API closely follows the AIM Schema (AIM XSD). Each class in the object model provides Set and Get methods for every attribute in the corresponding AIM Schema class. You can download the source code of the API from [5].

2. Methods

By using the API, the users can create new AIM XML files and edit any existing AIM XML files which can be current or older version of the AIM. The XML files can be saved to local file system or remote XML database. Additionally, the users are able to query one or set of AIM annotations from the server by using predefined functions. Structure of the API is shown in Figure 2.

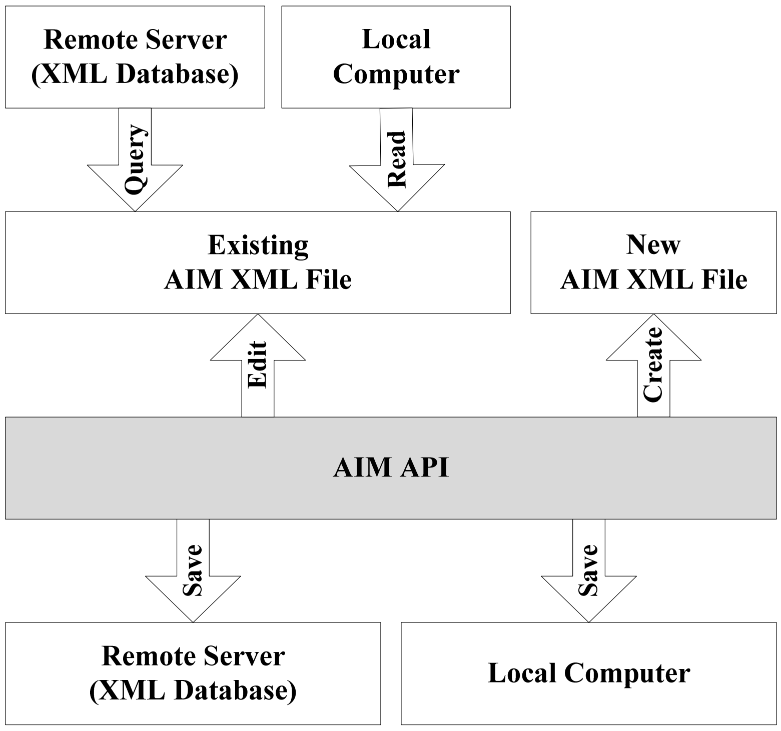

Figure 2 – Structure of the API

Since the AIM project started to develop, the version of AIM XSD have been upgraded (e.g., TCGA, 3.0 and 4.0) and each of them has some differences. AIM users not only have to understand the differences, but also they may have to update their codes for each new release of AIM XSD. Besides, in practice, source code modification may increase the error occurrence percentage of AIM users' projects and get their times.

For these reasons reading (parsing) any version AIM XML file is an important capability for the API. To do that, instead of creating individual AIM API for each AIM version, we have updated our API to be able to read (parse) any version of AIM XML file. Usage of the API is same for each AIM version, which means that the API users do not need to change their codes or algorithms depend on the AIM version. The API automatically detects the version of any given AIM XML file and creates required Java class instances. Then, users can read or edit the AIM XML file and save it in current AIM version.

On the other hand, to be able to analysis AIM annotations with Semantic Web technologies, the annotations must be converted to AIM ontology instances. So, we have updated the API with respect of this necessity. API users can save their AIM annotations into any existing or new AIM OWL (Web Ontology Language) [8] ontology file. While Figure 3 shows syntax of a sample AIM `Person` instance in AIM XML, Figure 4 shows RDF syntax of the same AIM `Person` instance in an AIM OWL file.

```
<Person birthDate="1953-01-01T00:00:00" sex="F" cagridId="0"
id="P481" name="1.3.6.1.4.1.9328.0022" />
```

Figure 3 – Sample Person in AIM XML

```
<AIM:Person rdf:ID="Person_P481">
  <AIM:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
    1953-01-01</AIM:birthDate>
  <AIM:sex rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    F</AIM:sex>
  <AIM:cagridId rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
    0</AIM:cagridId>
  <AIM:id rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    P481</AIM:id>
  <AIM:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   1.3.6.1.4.1.9328.0022</AIM:name>
</AIM:Person>
```

Figure 4 – Sample Person in AIM Ontology

Java Overview

We use Java programming language to develop the AIM API. Java is an Object oriented application programming language developed by Sun Microsystems. Java is a very powerful general-purpose programming language. Due to its versatility, it is a platform independent language, be it a hardware platform or any operating system.

XML Overview

XML stands for Extensible Markup Language and was defined 1998 by the World Wide Web Consortium (W3C). XML is designed to transport and store data. A XML document contains set of element; each element has a start tag, content and an end tag. A XML document must have exactly one root element, e.g. one tag which encloses the remaining tags. XML is key sensitive so it makes a difference between capital and non-capital letters. A XML file is required to be well-formatted. Well-formed XML must apply to the following conditions:

- A XML document always starts with a prolog which describes XML file. This prolog can be minimal, e.g. `<?xml version="1.0"?>`
- Every tag has a closing tag.
- All tags are completely nested.

A XML file is valid if it is well-formed and if it is contains a link to a XML schema (XSD) and is valid according to the schema.

XSD Overview

The XML Schema definition language (XSD) enables you to define the structure (elements and attributes) and data types for XML documents. A XML schema describes the coarse shape of the XML document such as what fields an element can contain, which sub elements it can contain, ordering of tags in the document etc. It can also describe the values that can be placed into any element or attribute. While, a well-formed XML document is one that satisfies the usual rules of XML, a valid document is one that is well-formed and that satisfies a schema.

Java XML Overview

Java contains several methods to access XML. The most popular methods to parse XML files with Java are using DOM API (Document Object Model) and SAX (Simple API for XML). In DOM, user can access the XML document over an object tree. DOM can be used to read and write XML files. On the other hand, SAX provides sequential reading of XML files. SAX can only read XML documents.

Details of the API

The AIM API consists of two main Java class groups. In the first group (package name is *edu.stanford.hakan.aim3api.base*), the classes model the AIM structure. In other words, their name and attributes closely follow the AIM Schema (e.g., `ImageAnnotation`, `DICOMImageReference`, `Person` etc.) and each class provides Set and Get methods for every attribute in the corresponding AIM Schema class. On the other hand, the second group (package name is *edu.stanford.hakan.aim3api.usage*) contains five Java classes to perform objective operations of AIM annotations such as *save*, *validate*, *read* etc. The common point of these classes is all methods belong to them are static Java methods. Thus, API users can use them without create any Java class instances. Brief description of the each class is given as follows,

`AnnotationBuilder` is a Java class which is responsible about save operations of the AIM annotations. API users can save their AIM `ImageAnnotation` instances to both local file system (Figure 5) and a XML database (Figure 6). During the save operations, user must specify the AIM XSD (*PathXSD*) to validate if the `ImageAnnotation` instance is valid or not. If the validation failed, the instance cannot be saved. Furthermore, before sending the `ImageAnnotation` instance to the server, the API checks if the annotation is already exist on the server. Because we don't want to insert more than one `ImageAnnotation` instance which has same *UniqueIdentifier* value.

```
//*** Saving an ImageAnnotaion (iAnnotation) to hard drive
AnnotationBuilder.saveToFile(iAnnotation, PathXML, PathXSD);
```

Figure 5 – Saving an `ImageAnnotation` to Local File System

```
//*** Saving an ImageAnnotaion (iAnnotation) to the server
AnnotationBuilder. saveToServer(iAnnotation, serverUrlUpload,
serverUrlDownload, namespace, collection, PathXSD);
```

Figure 6 - Saving an `ImageAnnotation` to Remote Server

`AnnotationGetter` is another important Java class of the API which is used to get any `ImageAnnotation` instance from an AIM XML file (Figure 7) or remote server (Figure 8). Methods of the class return one or list of `ImageAnnotation` instance. The methods first parse the AIM XML and then create related AIM API class instances to compose the `ImageAnnotation` instance. Before the parsing process, the methods check the AIM XML if it is valid according to the AIM XSD. If not, an informative Java exception is thrown and users can see the problem(s).

```
//*** Getting an ImageAnnotaion (iAnnotation) from hard drive
ImageAnnotation iAnnotation  =
AnnotationGetter.getImageAnnotationFromFile(PathXML, PathXSD);
```

Figure 7 – Getting an `ImageAnnotation` from Local File System

```
//*** Getting List of ImageAnnotaion class instance from the server
List<ImageAnnotation> listImageAnnotation =
AnnotationGetter.getImageAnnotationsFromServerByNameContains(serverURL,
namespace, collection, name, PathXSD);
```

Figure 8 – Getting List of `ImageAnnotation` from Remote Server

`AnnotationValidator` is responsible to validate any given AIM XML. The XML may come from a file located on the hard drive or XML database. In all conditions, the XML must validate according to AIM XSD. This validation process is very important to provide consistency between AIM annotations. Sample usage of the class is given in Figure 9. The method namely *ValidateXML* gets two parameters as path of AIM XML file and AIM XSD file and returns boolean (true or false) as result value.

```
//*** Validate AIM XML based on AIM XSD
Boolean valRes = ValidateXML(PathXML, PathXSD);
```

Figure 9 – Sample Validation

`AnnotationConverter` converts older AIM XML to current version. It contains set of conversation methods for AIM Schema classes, such as *UserTCGA2V3*, *imagingObservationTCGA2V3* etc. The API users do not use this class directly. While the API starts to parse any AIM XML, it detects the version of AIM annotation first and calls required methods automatically. So, the user does not need to carry about version of the AIM XML file.

`AnnotationExtender` is used to extend usage of the API depends on the its user's requests. In other words, AIM API users can request additional methods to simplify usage of the API in their projects. Sample additional method is shown in Figure 10. The method adds `Calculation` instance(s) to any given `ImageAnnotation` instance denotes as *iAnnotatio*n. Where *featureValue* is a double array, *featureString* is a string array and *featureVersion* is a double value. Finally the method returns updated `ImageAnnotation` instance as return value.

```
//*** Validate AIM XML based on AIM XSD
ImageAnnotation iAnno = AnnotationExtender.addFeature(iAnnotation,
featureValue, featureString, featureVersion);
```

Figure 10 – Sample Additional Method

3. Results

Who will use the API?

Brief information about list of the projects/researchers using AIM API can be expressed with their possible references…

Sample usage of the API

In this sections sample usage of the API is given to annotate a lesion on the mammogram. `ImageAnnotation` class of AIM is used to annotate images and the class is inherited from `Annotation` abstract class which captures general description of the AIM annotation such as *name*, *version* etc. Figure 11 shows creation of an `ImageAnnotation` instance with its required attributes.

```
//*** Creating an instance of ImageAnnotaion (iAnnotation)
ImageAnnotation iAnnotation = new ImageAnnotation();

//*** Setting required fields of the iAnnotation
iAnnotation.setCagridId(0);
iAnnotation.setAimVersion("AIM 3.0");
iAnnotation.setDateTime("2012-01-16T15:32:14");
iAnnotation.setName("Lesion 1");
iAnnotation.setUniqueIdentifier("ID-12345");
iAnnotation.setCodeValue("0");
iAnnotation.setCodeMeaning("Mass");
iAnnotation.setCodingSchemeDesignator("DCM");
iAnnotation.setComment("Sample Usage of AIM API");
```

Figure 11 – Creation of an AIM Image Annotation

Each `ImageAnnotation` instance must have a `Person` instance to state owner (Patient) of the annotation. Each `Person` instance must have *CagridId* is used to uniquely identify an instance of a class within a system implementing caGRID technology [6], *name* and *id* attributes. In addition to these attributes `Person` class may have *birthDate, sex* and *ethnicGroup*. Figure 12 shows creation of a sample `Person` instance by using AIM API and it is added to `ImageAnnotation` instance (*iAnnotation*).

```
//*** Creating a Person instance (person)
Person person = new Person();
person.setCagridId(0);
person.setName("Name Surname");
person.setId("112233");
person.setSex("F");
person.setBirthDate("1965-02-12");

//*** Adding Person to ImageAnnotation
iAnnotation.addPerson(person);
```

Figure 12 - Creation of an AIM Person

The image references classes specify the image or collection of images being annotated. AIM annotations may reference DICOM images or web images. In this paper we just give usage of `DICOMImageReference` class which simulates a subset of the DICOM information model. The class has one `ImageStudy` instance that has one or more `ImageSeries` instances which in turn have one or more `Image` instances. `ImageStudy` instance has *CagridId*, *UID* and *Date* and *Time*. `ImageSeries` instance has *CagridId* and *UID*. `Image` instance has *CagridId*, *SOP class UID*, *SOP instance UID*. Figure 12 shows creation of a `DICOMImageReference` instance and it is added to *iAnnotation*.

```
//*** Creating a DICOMImageReference instance (dicomImageReference)
DICOMImageReference dicomImageReference = new DICOMImageReference();
dicomImageReference.setCagridId(0);

//*** Creating a ImageStudy instance (imageStudy)
ImageStudy imageStudy = new ImageStudy();
imageStudy.setCagridId(0);
imageStudy.setInstanceUID("LCC");
imageStudy.setStartDate("2012-01-16");
imageStudy.setStartTime("12:45:34");

//*** Creating a ImageSeries instance (imageSeries)
ImageSeries imageSeries = new ImageSeries();
imageSeries.setCagridId(0);
imageSeries.setInstanceUID("LCC");

//*** Creating a Image instance (image)
Image image = new Image();
image.setCagridId(0);
image.setSopClassUID("112233");
image.setSopInstanceUID("445566");

//*** Adding Image to ImageSeries
imageSeries.addImage(image);
//*** Adding ImageSeries to ImageStudy
imageStudy.setImageSeries(imageSeries);
//*** Adding ImageStudy to ImageReference
dicomImageReference.setImageStudy(imageStudy);
//*** Adding ImageReference to ImageAnnotation
iAnnotation.addImageReference(dicomImageReference);
```

Figure 13 - Adding DICOM Image's information to the Image Annotation

`ImagingObservation` class is used for the description of “things” in the image. For example in mammography it can be “mass,” “calcification,” etc. `ImagingObservation` instance can be associated with one or more `ImagingObservationCharacteristic` instance which descriptors of `ImagingObservation`.

According to ACR’s (American College of Radiologists) BI-RADS (Breast Imaging Reporting and Data System) mammography atlas [7] each mass has three attributes; *shape, margin* and *density*. Furthermore, each attribute has its set of allowed values. For example, mass shape can be *round*, *lobular*, *oval* or *irregular*. Figure 14 shows a mass has *irregular* shape, *spiculated* margin and *high* density in the left breast of the patient with its boundary and representation of the mass in AIM structure. Additionally, Figure 15 shows usage of AIM API to add the annotation of the mass into *iAnnotation*.

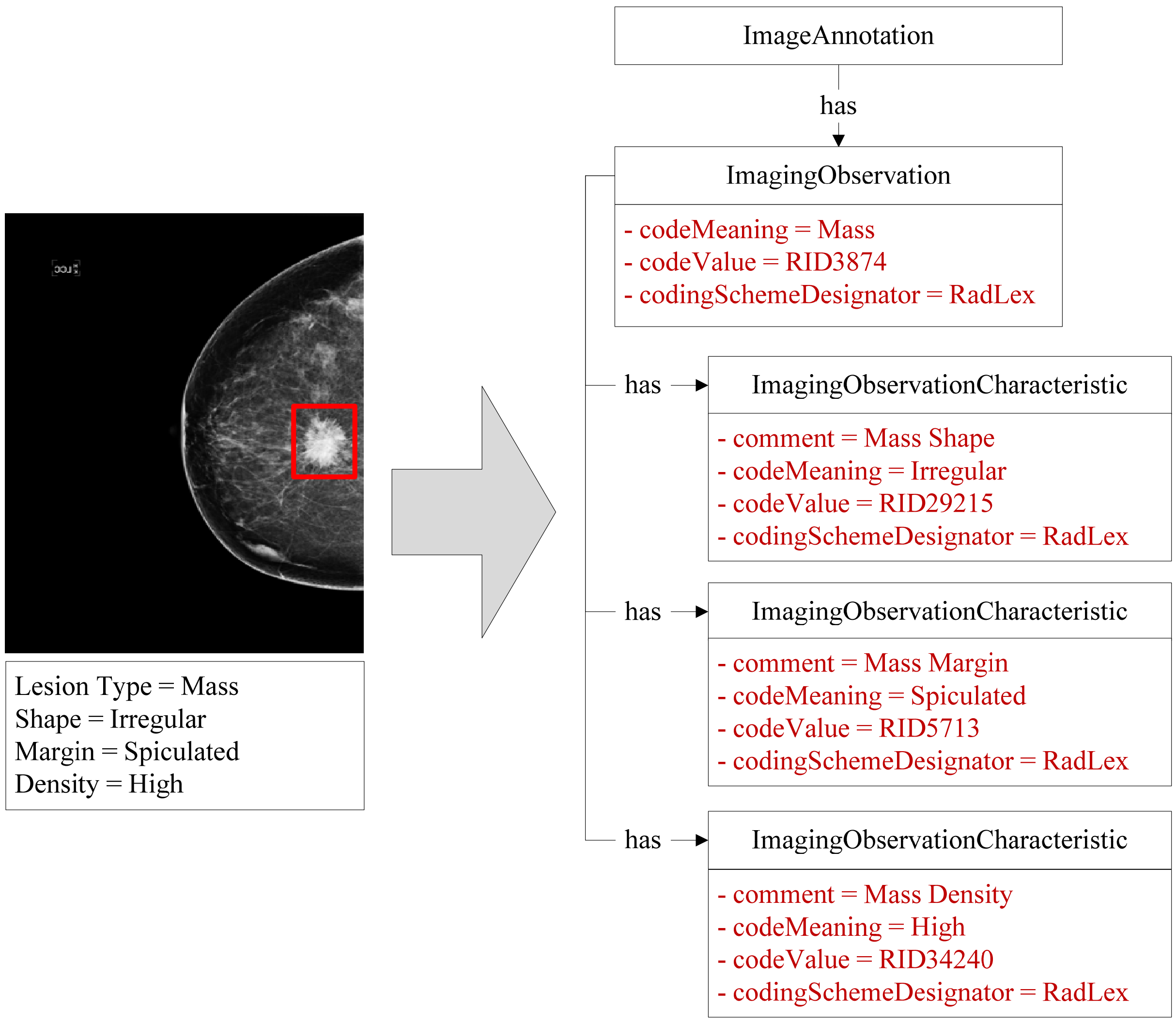

Figure 14 – Sample Breast Lesion and its AIM Representation

```
//*** Creating an instance of IO class (io)
ImagingObservation io;
io = new ImagingObservation();
io.setCagridId(0);
io.setCodeMeaning("Mass");
io.setCodeValue("RID3874");
io.setCodingSchemeDesignator("RadLex");

//*** Creating an instance of IOA (ioaShape) for Mass Shape
ImagingObservationCharacteristic ioaShape;
ioaShape = new ImagingObservationCharacteristic();
ioaShape.setCagridId(0);
ioaShape.setComment("Mass Shape");
ioaShape.setCodeMeaning("Irregular");
ioaShape.setCodeValue("RID29215");
ioaShape.setCodingSchemeDesignator("RadLex");

//*** Creating an instance of IOA (ioaMargin) for Mass Margin
ImagingObservationCharacteristic ioaMargin;
ioaMargin = new ImagingObservationCharacteristic();
ioaMargin.setCagridId(0);
ioaMargin.setComment("Mass Margin");
ioaMargin.setCodeMeaning("Spiculated");
ioaMargin.setCodeValue("RID5713");
ioaMargin.setCodingSchemeDesignator("RadLex");

//*** Creating an instance of IOA (ioaDensity) for Mass Density
ImagingObservationCharacteristic ioaDensity;
ioaDensity = new ImagingObservationCharacteristic();
ioaDensity.setCagridId(0);
ioaDensity.setComment("Mass Density");
ioaDensity.setCodeMeaning("High");
ioaDensity.setCodeValue("RID34240");
ioaDensity.setCodingSchemeDesignator("RadLex");

//*** Adding all instances of IOA to the IO instance
io.addImagingObservationCharacteristic(ioaShape);
io.addImagingObservationCharacteristic(ioaMargin);
io.addImagingObservationCharacteristic(ioaDensity);

//*** Adding instance of IO class to the Image Annotation
iAnnotation.addImagingObservation(io);
```

Figure 15 - Creation of an AIM Imaging Observation

Finally, we save the *iAnnotation* to local file system by using `AnnotationBuilder` Java class. Sample usage of the class is shown Figure 16. We use its *saveToFile* method which creates AIM XML file to given path (*PathXML*). Where, *PathXSD* denotes path of the AIM XSD file which is use to validate target AIM XML file.

```
//*** Saving iAnnotation) to local file system
String PathXML = “C://sampleAimAnnotation.xml”;
String PathXSD = “C://AimV3.xsd”;
AnnotationBuilder.saveToFile(iAnnotation, PathXML, PathXSD);
```

Figure 16 – Saving an *iAnnotation* to Local File System

4. Conclusion and Future Works

In this paper, we present a Java API to simplify usage of the AIM for the researchers. Details and sample usage of the API is given. The API makes possible to read and create AIM XML files by the researchers even they don’t have detail information about XML technologies. The development process of the API will continue according to its users’ feedbacks. We hope that, the API will increase count of the projects using AIM.

5. Acknowledgements

This work is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under project number 107E217 and National Cancer Institute (NCI) through the cancer Biomedical Informatics Grid (caBIG) Imaging Workspace subcontract from Booz-Allen & Hamilton, Inc. 85983CBS43.

6. References

[1] Rubin DL, Mongkolwat P, Kleper V, Supekar K and Channin DS, Medical Imaging on the Semantic Web: Annotation and Image Markup. In: 2008 AAAI Spring Symposium Series, Semantic Scientific Knowledge Integration, Stanford University, 2008.

[2] Rubin DL, Supekar K, Mongkolwat P, Kleper V and Channin DS, Annotation and Image Markup: Accessing and Interoperating with the Semantic Content in Medical Imaging. IEEE Intelligent Systems 24(1): 57-65, 2009.

[3] Channin DS, Mongkolwat P, Kleper V, Sepukar K and Rubin DL, The caBIG Annotation and Image Markup Project. J Digit Imaging, 2009.

[4] AIM API Web Page, http://www.stanford.edu/group/qil/cgi-bin/mediawiki/index.php/AIM_API, 2011.

[5] AIM API Spurce Code, http://sourceforge.net/projects/aimapi/, 2011.

[6] The caGrid, https://cabig.nci.nih.gov/community/workspaces/Architecture/caGrid, 2012

[7] C. J. D'Orsi, L. W. Bassett, and W. A. Berg, "BI-RADS: Mammography," in Breast Imaging Reporting and Data System: ACR BI-RADS – Breast Imaging Atlas, 4th ed., C. J. D'Orsi, E. B. Mendelson, and D. M. Ikeda, Eds. Reston, VA: American College of Radiology, 2003.

[8] OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/, 2012