RC23176 (W0311-170) November 24, 2003
Computer Science

IBM Research Report

An Effective, Java-Friendly Interface to the CAS

Marshall Schor
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

An Effective, Java-Friendly interface to the CAS

Introduction

A paper in the IBM the Systems Journal [CAS article in Systems Journal] describes the Common Analysis System (CAS) and gives motivations for it. We extend these motivations further, and describe an approach to working with the CAS from Java that is effective in many dimensions including type safety, maintainability, readability, performance, and composablility.

We call this approach the JCas, short for the Java interface to the CAS. The CAS, as part of UIMA (see [UIMA article in Systems Journal]), is not language specific; we have both a Java version and a C++ version that interoperate – in that one part of an analysis may be done by an annotator executing in one language/environment, and the results transferred to an annotator executing in the other environment.

The CAS paper focuses on the functionality of the CAS; this paper focuses on an effective Java interface to the CAS and compares it with other contemporary approaches for user-friendly interfaces, including Visual Basic [Visual Basic] and the Eclipse Modeling Framework (EMF) [EMF].

This paper is not self-contained; it presumes a previous understanding of the CAS system and UIMA, which can be obtained from the references.

Motivations

In addition to the motivations described in the CAS paper (Component Assembly, supporting Data-driven inference to aid in component assembly and interoperability, and an efficient serialization/marshalling capability), we observe that a significant success factor in delivering new platforms/frameworks like UIMA is **ease-of-learning** for the target intended audience. We expect most adopters of UIMA will use the Java version. In early prototypes, we observed users implementing their own ad-hoc approaches to making the CAS functions available in a familiar Java form.

XML is used as a declarative specification of the CAS types that an annotator works with. A second observation from the local user community was feedback that although XML might be a good way for computers to interchange information and meta-data, it was a continual source of difficulty for developers.

This led to a second goal for JCas – that of **having a “developer-friendly” specification approach that would be easy to read and write (by a human),** in a syntax that was familiar to Java programmers, and which could be used to generate the required XML, whenever changes occurred.

The CAS implementation has made many specific choices. Because we are early in the exploration of alternatives for some of the details of the architecture, it may be useful to **experiment with alternative implementations**. This is another motivation for having the JCas, because it is a layer on top of the base CAS implementation, and allows for this experimentation without source code re-writing. It architects an interface for the CAS that allows alternative implementations to be used underneath.

All of this layering could be objectionable if it caused extra overhead. A final motivation was to explore how to **provide a Java-familiar**

interface to the CAS while not introducing additional overhead. In fact, the current implementation allows a more efficient use of the CAS – which we will describe in the section on type safety, following.

The JCas interface to CAS objects

The Basic CAS interface

There is a common API supported in both C++ and Java for accessing the CAS directly. It involves obtaining handles for all types and features (a feature is a field within a type), and then using those handles in operations to create new type instances and get/set the fields within those types.

Typical code using this basic interface first declares variables to hold handles to the type and features of that type. Next, code that sets these via CAS methods that extract meta-data needs to run at some initialization point in the user's code. Once initialized, these handles are then used in getting and setting features.

Using common naming conventions to make a friendlier interface

Both Visual Basic, Java Beans [Java Beans], and EMF, among many others, adopt a convention of automatically-generated names that follow conventions. We adopt the same conventions: each field within a type named "xyz" will have generated methods named getXyz and setXyz, to read (get) and write (set) the field "xyz"; these methods are specific to the type. Of the various methods in the marketplace that have been tried to ease the learning curve, this approach seems to have gained a large following. Informal feedback from users attests to its ease-of-use and rapid learning curve.

We create new objects by using either the common "new" operator in Java or a common Java form for object creation: the create() method on a type-specific "factory".

Here's the same code fragment above, done with this syntax:

```
/* Create a new "Token" object */
Token aToken = new Token(jcas);
/* or - an alternate form if Token is an
Interface, not a Class */
Token aToken =
Token.factory.create(jcas);
/* set a features */
aToken.setTokenType = 1;
```

Because UIMA supports the notions of multiple CASs at runtime, the "jcas" parameter selects which instance of a CAS the token should be created in. We currently are using the "new" operator approach – as it is more familiar to Java programmers; however, for future extensibility, we will most likely be shifting to the "Interface" style which will require the "factory" approach.

By adopting this widely used standard convention for getting and setting fields, we expect to benefit from the form's familiarity, as well as from the same kind of tooling developed around this approach.

Specifying the types

The Java classes that define these types all have a similar structure. The declarative information about CAS types allows these classes to be automatically generated; this removes a potential source of error. EMF uses this same approach.

When classes are generated, what are they generated from? What is the specification input? In our design we have 2 sources. One is optimized for developer creation and maintenance; the other allows generation of the classes directly from a loaded CAS type system via reflection.

The CAS Type Specification file

The type specification for JCas is contained in a .cts file (.cts – the file type, meaning CAS Type Specification). The syntax of this file is modeled closely after Java syntax. But it isn't Java. There is an additional build step which converts it into Java classes. It contains all the information needed by the CAS type system. In designing this file format, we strove for something that would be notationally compact, do maximal factoring of things like name-space prefixes, and be easy to learn, easy to remember the meaning (easy to read), and easy to maintain. We use this file as input to a program which will generate the Java Class files that implement the interfaces, and also generate the XML type specification needed by UIMA. Having a generator reduces the sources of errors – the XML type specification is guaranteed to be in sync with the generated Java classes.

Human Readable Type Specifications

Here's a sample of a .cts file:

```
/** CAS type definition for ParseFrame
 * @author someAuthor@a.b.com **/

casNameSpace com.b.a.projectName;
// multiple casType specs follow -
// each looks like this:

casType ParseFrame extends Annotation {
   Integer     seqNo; //unique id
   Word        headWord;
   StringArray features;
   FSArray<ParseFrame> lMods;
   String      slotName;
   ParseFrame  parent;//null for top node
}
```

Some comments on the example: The feature types are either built-in to the CAS system (e.g. Integer, StringArray, FSArray, String, Annotation) or refer to other defined CAS types. The "extends" keyword specifies the super-type, with identical semantics to Java's use of "extends". The FSArray built-in type is followed by a specification <ParseFrame> which uses the coming syntax in Java 1.5 for Generics [Java 1.5 Generics], and has the same semantic meaning. (Although we support generic-like arrays, we do not require the use of a Java level that supports templates, for instance, Java 1.5.)

Wherever possible, we follow Java conventions. The casTypes look like Java Class declarations, except that they have no methods, only fields (CAS Features). The range-types of those fields are the allowed types in the CAS system, instead of being Java types. Comments follow Java conventions; javadoc [JavaDoc] comments are supported.

For the same reasons that Java implemented packages and imports, we use a casNameSpace statement to specify both the name space for the type names in the CAS, and the package name for the generated Java Classes. Import statements are carried over into the generated Java code to allow referring to types in other name spaces (packages) while factoring out the package name, resulting in easier-to-read and maintain source code.

The result is a notationally compact form of the type specification, which can generate both the Java classes implementing the JCas getter/setter interfaces as well as the type description XML that the UIMA architecture uses. As would be expected, this syntax-rich notation is much more compact than its corresponding XML specification, and much easier to read and maintain by developers.

Alternative specification approaches

EMF has a very similar approach to data modeling. It also provides for a generator to go from a specification to Java classes which have getter/setter methods. The input sources for EMF models can be a UML data model, or a Java interface specification annotated with additional information contained in comments using Javadoc-like tags, or XML forms of the specifications. The XML spec form is very similar to the CAS XML type specification form, so the capabilities here are similar.

The UML data model approach for the CAS could be done, but UML data modeling supports a much broader set of capabilities than are currently architected in the CAS. For example, UML data modeling allows modeling of non-CAS concepts, such as two-way links among instances, containment, etc.

The Java interface specification annotated with additional information is a notation that is not as compact and easy to read as the .cts notation. However, this form of adding additional (meta) information to Java classes, information needed by other processes, is becoming more common; Java 1.5 itself uses more of this approach. For now, we feel the notational compactness may aid wider adoption because it may be easier to learn and use.

CAS Arrays

In addition to getters and setters for fields, we extend the getter/setter capability to include getting/setting an element of a CAS Array, when the field is a CAS array.

Here is an example of the use of this:

```
/* get a value from an array */
aToken.getTokenFeature(myIndex);
/* set a value into an array */
aToken.setTokenFeature(
    myIndex+1, valueToSet);
```

Having these kinds of additional getters/setters allows for more efficient implementation of these functions; in particular, we avoid creating extra temporary Java Array objects whose only purpose is to allow getting or setting an element within it.

Type Checking

A cornerstone of modern language and notations is the concept of type checking – both at run-time and compile time.

Stronger-than-Java type checking

The normal type checking done by languages such as Java is being extended in Java 1.5 via the new template mechanism. This mechanism allows specifying for collections, the class of the objects in the collection. A major value claimed for this is the elimination of repetitive casting operations during accessing; these are in a sense factored out into the collection type specification, resulting in cleaner, easier to read, understand, and maintain code.

We add this same kind of strengthening of the type system for collections to CAS arrays, by allowing the developer to specify for arrays of Objects the type of the object in the array. This type information is specified using the same syntax proposed for this function in Java 1.5; it is used for both compile-time checking the arguments of values passed to "setter" functions, as well as for eliminating the need to cast results retrieved from these CAS arrays.

EMF has a similar approach to type-safety-strengthening.

Java Load Time operations

Java design supports operations done at class load time, such as initialization of static fields. When these fields are declared "final", the Java JIT compilers can use this information to do optimizations that compilers do when variables are treatable as constants. For instance, it can depend on the fact that the variables will never be modified and cache their values in registers. We take advantage of this to move most of the run-time checking (done by the basic CAS interface) so that it occurs at load time and results in constants the Java JIT compiler can benefit from optimize around.

When the CAS is initialized, the corresponding JCas classes are loaded. During this loading process, checks and initialization are done to validate the dynamically created CAS type system matches the JCas class definitions. Java "final" constants are initialized based on the instantiated CAS type system, and these constants are later compiled into the JIT generated code for the class.

We were able to move most of the run-time type checking previously done in the basic CAS interface into this load-time initialization, which works with the compile-time Java type checking. As a consequence, the JCas interface to the CAS runs significantly faster than the basic CAS interface design, with full type checking.

Using .cts files

To use the .cts files in an UIMA application, a developer runs (typically, as a part of his build process) a utility called **JCasGen** which reads the .cts file and creates corresponding Java classes that implement the get/set interfaces. These classes are added to the classes the developer is coding, as part of his application. The JCasGen utility also produces the XML specifications for the defined types, needed by UIMA when applications are assembled and deployed.

Type Augmentation at "Assembly time"

The companion CAS paper describes a scenario where one annotator might want to add additional features to the output of an existing annotator, by adding a field. This capability is not natively supported in Java. One approach to doing this has been to extend a base class. Here's the scenario:

Let's imagine Annotator A outputs "Token" annotations. Annotator B wants to augment this Token annotation with an additional field, perhaps a part-of-speech tag. In the basic CAS support, Annotator B can define the Token type with this additional field. At Assembly / Load time, all type descriptors are read for all annotators that will be sharing a CAS, and their type definitions merged. The resulting augmented type is used at run time. (Assembly time is an optional step that allows the computations otherwise done at load time, every time an application is loaded, to be done once, so loading can be faster).

The common approach to this kind of type augmentation in Java is subclassing. In this scenario – Annotator B would define a subtype of Annotator A's Token type, called, for instance, TokenB; the body of TokenB would be the additional Part-of-Speech field. In practice, this doesn't work very effectively, because Annotator A runs with its type definition, and when Annotator B starts, it has to copy the Annotator A's Token types into instances of its type (which have the extra field).

Other approaches to extending types are found in the "Adapter" pattern described in the EMF book. In this approach, an instance of type A is "adapted" to have another set of methods and fields. These methods and fields actually exist in another Java object instance, which is associated with the original instance, frequently via a hash table. This approach is used also in the Eclipse [Eclipse] technology, for connecting data models with UI data.

To emulate the basic CAS capability while keeping the additional type checking possibilities that come with the "subclassing" approach, we implement an Assembly / Run-time equivalent for adding fields to a type, in Java. In this approach, Annotator B would use subclassing, but things would be done at Assembly / Load time to allow types of Annotator A's Token to be "downcast" into types of Annotator B's.

This capability is only installed for types marked "downcastable", since Annotators commonly subclass annotations with no expectation of those types needing to be downcastable. An example is the type "Annotation" which defines only two integer fields, the beginning and the end of the annotation, and which is used by CAS type developers as a supertype of their particular annotation type.

When an Annotator designer wants to add a field to an existing type in a downcastable manner (to avoid the overhead of copying), they designate in the .cts file not only the type it extends, but also that it should be downcastable from that type. At assembly or load time, when the type specifications are gathered together, the JCas implementation arranges for the implementation of both the super and subtype to be that of the subtype; an upstream annotator, only knowing about the supertype, makes instances of a richer type structure but only accesses the fields it knows about. A subsequent downstream annotator can then down-cast the instances to that of the subtype, and add its information.

In Java, this is done using the Java interface mechanism, which allows alternative concrete implementations supporting a common interface definition.

Need for multiple-inheritance

Consider now an annotator A, and two down stream annotators B and C, each of which, independently, declare a type defined in A as a supertype of two different types in B and C, each marked downcastable. The implementation now must be the union of A, B and C. In the Java interface language, the implementation must implement A, B, and C. Fortunately, this works fine – Java interfaces support multiple inheritance.

Implementation at Assembly or Load time

To implement the a concrete class that has the union of these types, we choose an approach that merges the .cts files and generates and compiles the required Java implementation, at assembly time, when the Annotators that are to run together for a particular application, are specified. This can be also done at Load time, but it would involve invoking the Java compiler (which may not be available) on the generated classes at load time, and arranging for the class loader to find the generated implementation classes.

Comparison: JCas and EMF

EMF has become a cornerstone of WebSphere applications.   It has many of the same capabilities that the JCas brings to data modeling, but has a somewhat different set of goals it is trying to achieve. Both approaches have at their core the idea of generating Java code from "specifications". The JCas defines a compact, notationally convenient Java-like syntax, with maximal factoring for its input. EMF takes as input either comment-tagged Java interface sources, or UML data models. Both JCas and EMF can additionally take an XML specification as their input.

Data Model comparison

Basic data model

CAS Types include integers, floats, strings, references to other instances of CAS types, and arrays of these. EMF supports essentially all of the data types available in Java, plus types defined in EMF. The more restrictive approach taken in the CAS design allow for high-speed interoperability between Java and C++ implementations.

EMF capabilities not present in JCas

EMF has its roots in UML data modeling. It implements the UML concepts of bi-directional links, and containment (with an implicit inverse link). These result in the generated setter methods for the types implementing code that maintains the inverse relationships. JCas could do this too – but so far we haven't had the requirements to support these more complex kinds of data linkages for data in the CAS.

EMF supports a rich structure for keeping instances of the data being modeled in secondary storage, in a serialized form. It has support for "lazy" loading – avoiding reading in data until the references are actually followed, for references which cross package boundaries. Packages serve as containers of sorts for saved instances.

CAS as a container

The JCas/CAS is an in-core design, without any special focus on secondary storage. As a container, there is a richer support for accessing objects via indexes. A CAS instance itself serves as a container for object instances; the package system is orthogonal to this, and serves the same purpose as packages do in Java – that of being separable name spaces to avoid unwanted name collisions. A particular CAS could hold objects whose types are in many different packages.

Editors

The EMF generator concept is also applied to create a customizable, Eclipse-based editing environment for EMF type instances. The envisioned CAS use, being that it used as an inter-component communication vehicle, not as a persistent storage model, hasn't given rise (at least yet) to the need for this kind of function. We do have general viewers and editors for annotations, for instance, however, that are driven by the run-time meta-type-data that is the cornerstone of the CAS design. (EMF has a similar capability derived from similar run-time meta-type-data information, in addition to being able to generate a customized editor for particular sets of types).

JCas/CAS capabilities not in EMF

JCas/CAS supports a dual environment (Java and C++) capability, with very efficient marshalling of the CAS data between these environments.

The JCas/CAS design is based on a separation of concerns – envisioning a role for a code writer/developer, and separate roles for "assemblers" and people doing deployment, perhaps across multiple servers, running different language environments.

The JCas places functionality at "Assembly" or "load" time, moving, for instance, aspects of run-time type checking, in a way to couple this with Java compile-time type enforcement to have a highly efficient type safe runtime. This supports the downcastable capability for directly augmenting types at Assembly/Load time.

Experimental and Future Extensions

Future directions for JCas include Eclipse-based developer support, along the lines that EMF has already done.

The .cts specification allows the type specification to incorporate arbitrary Java code that can define additional fields and methods for the Java class implementation; these fields and methods are not part of the CAS itself, but are generated into the JCas class definition that is built for the CAS type. This extension can make the assembly process more manual, when multiple .cts files with perhaps conflicting arbitrary Java extensions need to be merged. We continue to evaluate the pros and cons of this capability. EMF likewise has a similar capability – the generated classes can be arbitrarily modified by the developer, and EMF is careful to preserve those modifications when regeneration occurs.

Some additional meta-data concepts being explored include marking data use as read-only – this could allow the UIMA component that orchestrates flow among annotators to multi-thread read-only annotators without worrying about locks.

We have prototyped versions of the JCas that extend the generator ideas to support a faster-than-Java Java – a scheme for accessing data in the CAS which doesn't actually use or produce

Java objects as a side effect. A common rule-of-thumb for Java has been the 1-10-100 rule for performance (1 = time to access a field in an object, 10 – time to call a method (that isn't inlined), 100 = time to create an object (including amortizing the Garbage Collection time); it may be interesting to have this alternative for very-high performance users of the CAS.

The availability of meta-data may allow semi or fully automatic adapting of annotators by mapping different accessing names into the same objects. For example, if two annotators should run sequentially, and the first one produces what the second one wants, only that the names chosen for the types are different, the Assembly operation can detect this, and (perhaps with human guidance) construct the mapping to bridge this. The existence of the JCas interface means that the annotator code would remain the same, only the JCas implementation of the classes would change to bridge the components. In this case, one implementation type implementing both interfaces would be defined.

Conclusion

The JCas interface to the CAS was designed to be very easy to use and learn. It uses a set of conventional styles familiar to Java programmers; the generator approach insures correct Java Classes and corresponding UIMA XML type specifications are produced from a common, easy-to-maintain source definition.

The JCas interface to the CAS is allowing a unique blend of compile-time, load-time, and run-time type checking that performs very well compared with the basic CAS interface. The ability in the CAS to support combining annotators where one adds fields to another's Type is supported in the JCas thru a mechanism labeled downcastable. JCas has different and somewhat more specific goals than the EMF approach, although the two share many aspects in common. JCas provides an architected layer that permits future experimentation with implementation alternatives, with minimal impact to existing Annotators.

Acknowledgements

The author benefited greatly from extended discussions regarding the CAS with David Ferrucci, Thilo Goetz, Thomas Hampp, and Adam Lally. Graeme Dixon and Herb Derby graciously helped the author achieve a more global and complete understanding of EMF and its significance. Alan Marwick gave several useful recommendations on the syntax of the .cts specification. Finally, the author thanks Arthur Ciccolo for his support and encouragement in this undertaking.

References

[Eclipse] http://www.eclipse.org

[EMF] http://www.eclipse.org/emf

[Visual Basic] http://msdn.microsoft.com/vbasic

[UIMA article in Systems Journal] D.Ferrucci and A.Lally, "Building an example application with UIMA," *IBM Systems Journal* **43**, Number 3, xxx-xxx (2004).

[CAS article in Systems Journal] T.Goetz and O.Suhre, "Design and Implementation of the UIMA Common Analysis System," *IBM Systems Journal* **43**, Number 3, xxx-xxx (2004).

[Java Beans] http://java.sun.com/products/javabeans/docs

[Java 1.5 Generics] http://jcp.org/aboutJava/communityprocess/review/jsr014

[JavaDoc] http://java.sun.com/j2se/javadoc