UIMAJ v3 Customized JCas classes and PEAR Support

Updated 7/2016

This page discusses ideas for supporting JCas customization, and multiple customizations possible due to PEAR classpath isolation with "customized" JCas cover classes.

The problem arises because with the Feature Structure data contained solely in a particular instance of a Java class, there must be an approach to have the differing contexts (different PEAR definitions of the customization of the JCas cover class) share and update this data. An instance produced from a PEAR needs to be able to be used outside of the PEAR, and vice-versa.

The current thinking for PEARs is to support a hierarchy of approaches, from completely general (but slower) where FSs are serialized / deserialized when crossing the PEAR boundary, to optimized approaches attempting to avoid the serialization / deserialization overhead.

A catalog of customizations observed in JCas implementations

Customizations are used for many things; here are some gleaned from some actual use cases:

- Implementations which keep Cas complex objects (e.g. FsLists) in parallel Java data structures (e.g. HashSets).
- Getters which return Java collection objects instead of declared type (e.g. type = FsList, object returned is an ArrayList<FeatureStructure> or a Set<FeatureStructure>)
- Additional "setter" forms. example: a setter for a float feature, taking a "Float" argument, and handling various NaN style including mapping "null" to one of them, then calling the underlying setter
- Additional methods that operate on multiple fields at once (perhaps keeping them in "sync")
- Override clone to do a deeper copy
- modify a getter - if null is the value, substitute a custom alternative
- Additional arbitrary methods that produce values via operations on features
- Additional constructors, taking more arguments, setting fields
- Additional fields (not in the type system, not serializable, etc.)
- custom tostring methods
- defining a custom compareTo method

Starting with customization, adding any merged type features

The merged type system (including PEARs) represents the common needed JCas cover class's fields and getters and setters. Merge this with any customization code while loading the cover class.

Implementation ideas here in diagram form: UV3 JCasClassLoading.

Handling multiple customizations of a UIMA Cas Type

Multiple customizations can arise when two or more independently developed components both implement customized JCas classes. This may happen in two ways:

1. An aggregate or custom application includes two (or more) components, and the classpath is set up in such a way that each component's customizations (which would be identically named Java Classes) are in the classpath. In this case, one of the customizations is ahead of the other in the classpath and will "win" in the class lookup.
2. An aggregate or custom application includes two (or more) components, one (or more) of which is a PEAR, and a type with multiple customizations occurs within the PEAR and outside of the PEAR. PEARs have their own classpath, and are intended to have their classpath override the containing aggregate's classpath. The "effective" classpath is switched when the PEAR component is entered, and switched back upon exit.

Alternatives for handling Aggregate case (1):

- Do the normal classpath resolution; one definition "wins", according to the rules of Java classpath searching. This is what is done in current UIMA, for this case.
- (Not done) ~~Do a full classpath search and locate **all** the definitions for a class, and do some kind of multiple definition resolution~~
  - ~~Throw an error, with a descriptive error messageAttempt to "merge" - this seems computationally hard to impossible in some cases.~~
    - ~~allow the person assembling the aggregate to resolve this by excluding all but one (somehow - not defined - maven has some capabilities here)~~

Alternatives for handling PEAR case (2):

- Base case: Assume the worst
  - Assume a successful type system merge
  - Make a class loader for the Pear.
  - At type system commit time for the Pear, use its class loader (and the parent class loader) to load any new JCas class definitions.
    - Type System references back into the loaded classes now need to be conditioned on the Pear environment, to pick the right definition among potential multiples
    - Any new Pear JCas class must become the cover class for any subtypes even if the subtype has a JCas class def in the outer context
  - JCas class instance creation for a Pear as a wrapper for the outer instance
    - inner shares the int[] and obj[], has same _typeImpl and _view refs
      - feature updates to one seen by the other, no need to serialize/deserialize
      - parallels the old v2 JCas idea of instance being a cover class to an underlying impl

- same JCasHashMap lookup from id value, if not found, then generate wrapper.

- Use the capabilities input/output: for the case where the particular class in the PEAR is **not** among the inputs or outputs, just isolate it:
  - switch and use the alternative definition, loaded under a separate class loader used for the PEAR.
  - ⚠️if an outside-the-pear Feature Structure holds a reference to this - could lead to class-cast exceptions
- Isolate it - switch and use the alternative definition
  - ⚠️This is what the current UIMA does; suffers from class-cast exception problems
- Throw an error
- Attempt to merge

❓A single common system-generated implementation + an augmentation mechanism for (multiple) customizations

At the high abstraction level: UIMA is providing

- An externalized representation of a type + features with single inheritance data model
- The idea that at pipe-line startup, components' individual type definitions are "merged" - allowing types to have the union of all features defined. This was done to make it easier to develop reusable components that could be independently developed and later assembled together.
- A CAS model where a big collection of these Feature Structures are passed among (separately developed) Annotator components.
- High efficiency (vs some other component models) for accessing this data, including the ability (in JCas) to "customize" the basic create and access fields capability

This idea is to separate the concepts of customizing the JCas cover classes as follows:

- At startup have the system generate from the merged type specifications (this includes types defined or augmented in PEARs) the common merged type system
- Have customization done via hooks to user code. The customization would need to specify two things: what to hook, and the hook itself. The hook itself would be a method to run. What to hook would be a specification (read by the framework at startup time). The kinds of things that could be hooked are chosen to emulate the current customization capability:
  - Creation of new instances
  - setting / getting of particular Features
- References to Feature Structures would reference the system generated one, which (if customization was active) in turn would call the appropriate hook functions.
- Multiple different Annotators could have different customizations (something not currently supported).
- "Boundaries" for where the customizations were active could be established (to mimic a PEAR going in and out of scope).

This may be very hard to implement. There are examples of new methods being added in a customization; these methods would need to be called from a base instance (don't know how to do that). Also, customization may add fields.

❓Copy some Feature Structures into / out of the PEAR

A PEAR represents a packaging and a classpath isolation. The PEAR may (for some of the shared types) have a different definition of the JCas cover class. If the difference is simply it has fewer features than the pipeline's fully merged definition, then it can use the main pipeline's definition (it will have extra methods, but those would just go unused). If it has a different implementation beyond that, then a lazy (when needed) copying of the main pipeline's Feature Structure into the Pear's instantiation would be done. Likewise, on exit from the PEAR, if the type is mentioned as being an output, and it was "copied" in, then it would be copied out.

- Advantages:
  - no overhead if PEAR doesn't have customized JCas types.
- Disadvantages:
  - copying of FS in two directions.
  - overhead of remembering what FS need copying-out at exit from PEAR

Needs:

- isStandardJCasClass() to test if the JCas is **not** customized

❓Merge all the custom definitions into one and use that everywhere.

Do some kind of "merge" operation among all definitions of JCas cover classes including those in contained PEARs, and use that one merged definition everywhere.

- Advantages: is most similar to what we have now
- Disadvantages: it's not always possible to find a merge that preserves all the original implementations. It might be very difficult to construct an appropriate merge algorithm, given the arbitrariness of the custom code.

👎Keep the system-generated code in one class, and use a different wrapping class for the customization

Split apart the system-generated (from the merged type system) JCas cover class and user customization, into different classes. The user customization class would wrap the system-generated one, and create both; all value setting/getting would be via forwarding methods.

Advantages:

- No merging logic is needed; it would allow dropping the merge facility (which is old, doesn't support Java 1.5 or later, etc.)
- The system could generate from the merged type system a cover class that supported all the fields, making full use of the type hierarchy. There would be no need to have external processes or procedures to insure that the cover class generated had the fully merged type system; this would be automatic. Projects could run JCasGen to get prototype cover classes - these would not be loaded but would serve to provide classes to have code compile against.
- Better management of customization vs system code due to their separation.

Disadvantages:

- This approach seems to break the type inheritance model (a custom class wrapping the system-generated one would not be in a Java type hierarchy. The normal way around this is to have "interfaces" in the hierarchy. However interfaces can be created with "new ...". I suppose we could change things to not rely on "new" (e.g., have a create() operator). But that would be a big change.
- It would require some kind of a migration utility, because this is not how users customized the generated classes.
- It would end up with one more re-direction for get/set operations (due to the wrapper), for customized classes. If no customization was needed, the generated class would be named with the official name and serve all uses of it.

Nesting: An "outermost" pipeline can nest 1 or more PEARs, which, in turn may nest one or more inner PEARs, etc. (Type merging is applied to all the type definitions, including those in the PEAR). Each inner PEAR JCas customization would be a 1-level wrapper of the system-generated class from the outer-most pipeline (not on its container if it was a PEAR).

Naming: JCas cover classes are named to match their UIMA type name. This enables users to write "new MyType()" where MyType is the UIMA type name.

- If a JCas cover class is not customized (anywhere in the pipeline, including inside PEAR files), we have the system generated class, and its expected name as it is now.
- If it is customized, the custom "wrapper" would carry the official name, so users would use it, and the system-generated class would need a new name (e.g. xxxx_UIMA_JCas_Generated.), which would "hide it" from normal access. A complete analysis of the pipeline running as an application in one JVM would be needed to find (including inside PEARs) which UIMA types had customization (anywhere, including even if in just one PEAR). Those types would need the alternate naming protocol.

👎Keep the system-generated code in one class, and do the customization in a class "extension"

Another way to split apart the system-generated from the customization: have the customization "extend" rather than wrap the system-generated one.

Advantages

- Type hierarchy / inheritance works (sort of).

Disadvantages

- If you extend a class to cusotmize it, and some classes in the parent chain are also extended, your extended class misses those customizations. (This is avoided if you instead "merge").