Summary of research on modularizing Demeter class dictionaries

Or the story of how my eyes were bigger than my stomach

Will Guaraldi and Karl Lieberherr
{guaraldi,lieber}@ccs.neu.edu
College of Computer and Information Science
Northeastern University
360 Huntington Avenue 202 WVH
Boston, Massachusetts 02115 USA

ABSTRACT

For the Spring 2007 semester, I worked on a research project focusing on ways to modularize Demeter class dictionaries. This paper summarizes my research in this area.

Note that this paper is less of a formal technical report and more of a loosely organized collection of observations, research discoveries, and thoughts on possible future directions. This paper serves as a summary of the research I did as well as a possible springboard for someone else to continue research in this area.

This paper assumes you are familiar with DJ [5], Demeter [3], DemeterJ [4], DAJ [2], class dictionaries, class graphs, and strategies.

Keywords

Demeter, class dictionaries, modularizing class dictionaries

1. MOTIVATIONS

The original motivation I had for researching modularizing Demeter class dictionaries came from working on a project at the end of the Fall 2006 semester of CSG260: Advanced Software Engineering. We built a verifier that took CSP input files in a variety of formats and a solver output file and verified that the solver's solution specified in the output file solved the CSP specified in the input file.

The CSP input file and the output file formats were already specified in class dictionaries and used by other projects. We wanted to use the existing class dictionary files in our verifier allowing us to parse the input files that they were parsing. We wanted to write a single class dictionary file that included or imported the other class dictionary files treating them as libraries.

DemeterJ (and friends) have no ability to import/include class dictionaries. We could have created a single class dictionary file and copy and pasted the contents of all the class dictionary files we wanted into it, except there were naming conflicts between the class dictionaries.

We had to create a separate DemeterJ project for each class dictionary file we wanted to work with. Each project had to be in a separate Java class package because of name space collisions between the class dictionary files. We had to make changes to the class dictionary files and their behavior files because they weren't set up to be in different Java packages and we had to merge them into a Java package hierarchy. We had to add additional behavior code to all these projects that we needed for our verifier, which turned out to be a lot of code duplication. The end result is that we had local versions of the code the other groups were producing that had changes unique to our project and any time they made changes we'd have to manually merge those changes into our local versions.

Ideally we would have been able to create a single class dictionary file that imported the other class dictionary files in different name spaces all within a single DemeterJ project and defined our behavior in our behavior files based on our class dictionary.

The second motivational anecdote was that Ahmed was working on a class dictionary for an XML-based CSP input file. This was for an undergraduate software development class. Over the span of a few weeks, the undergraduate class requested a series of changes to the input format to make it easier to work with. For every change they made to the format, Ahmed would go through great pains to adjust the class dictionary which was 135 lines long and growing. He suggested that it would have been easier if he could break the monolithic class dictionary into a series of smaller class dictionaries focusing on specific bits of the XML-based CSP format.

Lastly, Theo was thinking about updating a class dictionary of the Java grammar from version Java 1.1 to Java 1.5. However the size of the class dictionary for the Java 1.1 grammar was huge and it looked much easier to start from scratch than figure out what changes needed to be made to the monolithic class dictionary file.

It's important to notice that class dictionaries are only one piece of adaptive-programming. Class dictionaries don't exist in a vacuum–they're often accompanied by behavior files and strategies which refer to the structure specified by the class dictionary. Because of this, modularization of class dictionaries isn't simply a mission to find a way to include and refer to other files; modularization also affects the behavior written against the structure. Modularizing class dictionaries walks a fine line between modules in structure documents (XML Schema, XML DTDs, ...) and modules in programming languages (SML, ...).

These motivating anecdotes suggest the following possible list of requirements from a class dictionary modularization approach.

1. It should allow us to break up a single class dictionary into a series of files.
2. Classes from imported/included class dictionaries need unique names so that classes with the same name from two different imported/included class dictionaries don't clash. This could be done with name spaces or α-conversion. Additionally, since classes are referred to in behavior files and strategies any transformation of class names needs to be deterministic as well as unique.
3. It might be useful to be able to extend classes that are imported/included from another class dictionary.
4. Importing/including class dictionaries can't break strategies and behavior files.
5. We'll need to verify that the constraints for strategies and class dictionaries are maintained. (Note: Demeter Interfaces adds additional constraints for strategies.)

2. CONSTRAINTS

2.1 Class dictionaries

When modularizing class dictionaries, we need to make sure we're not violating any of the constraints for class dictionaries. These constraints are specified in the DemeterJ user manual [4]:

1. **inductiveness axiom** - Classes in the class dictionary can't be inherently circular. DemeterJ creates a parser that takes a stream of characters and generates an object graph based on the class dictionary. If the class dictionary is not inductive, then the parser won't work right.
2. **cycle-free axiom** - A class can't inherit from itself.
3. **unique label axiom** - Classes can't have more than one edge with the same label.
4. **alternatives** - Alternatives of classes have to be classes–they can't be strings or other terminals.
5. **class names** - Class names must start with a capital letter.

Since class dictionaries are specified with LL(1) grammars, the new class dictionary which is a result of modularization must also be LL(1).

2.2 Strategies

When working out modularization possibilities with class dictionaries, it's important to think about how this affects strategies.

1. **uniqueness** - If the strategy specifies a unique PathSet in the class dictionary that specifies graph A, then it must specify a unique PathSet in the class dictionary that results from modularization.
2. Demeter Interfaces allow you to specify constraints on strategies [9]. Modularizing class dictionaries must not break these constraints either. I'll talk more about Demeter Interfaces later.

3. RESEARCH

3.1 XML Schema

XML Schema allows for importing of other schemas into schema files [7]. When schemas are imported, they are given a namespace identifier and this identifier allows you to unambiguously refer to types defined in that schema. This allows the creation of type libraries that are reusable across multiple schemas. For example:

```
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="stringlist"
               type="xsd:string" />
</xsd:schema>
```

Additionally, XML Schema allows for creating complexTypes that extend types defined in the same file or in imported schema files. Extensions can extend or restrict the extended type.

If we added a way for importing other class dictionaries into a sepcified name space, then we could treat class dictionaries as modules. This allows us to modularize class dictionaries and refer to classes we've imported without name space collisions. For example:

```
import "./uginput/program.cd" as ugi

UgiInput = ugi:instance .
```

However class dictionaries aren't alone–they usually come with behavior files and strategies. We could couple class dictionaries with their behavior files and strategies into a module. When we import a class dictionary into another class dictionary, we'd really be importing the entire module.

DemeterJ has the idea of a module in a DemeterJ project. Perhaps this structure could be used for import purposes, too.

For example, we create a class dictionary of bus routes for a city. The strategies for this class dictionary would specify PathSets for finding the locations for a specific busroute. We want to look at all the bus routes for all the cities on the east coast. As such we create a class dictionary to model cities. It imports the bus route class dictionary into the `br` name space and refers to the classes in the bus route class dictionary that way. Additionally, the city class dictionary has a strategy that gets a list of all the bus routes with `from Nation to br:Location`.

One interesting thing to mention here is that DAJ uses ANTLR for parsing. ANTLR can do grammar inheritance [1]. It's possible that DAJ (DemeterJ uses JavaCC) could do a pre-processing step that pulls all the class dictionary files into a super grammar and uses ANTLR's grammar inheritance. This idea was never fleshed out to see if it was useful or not.

3.2 Demeter Interfaces

Demeter Interfaces [9] provides an interface abstraction layer between the structure of the program (class dictionary/class graph) and the behavior of the program (strategies, traversals, and behavior files). Demeter Interfaces allow you to change the structure with a compile-time check as to whether the structure satisfies the constraints of the interface.

To implement an interface you have to create a class dictionary and then map things specified in the interface to things in the class dictionary. There are two primary cases of mappings:

1. **use** <*cd-class*> **as** <*interface-class*>
   This maps a class in the class dictionary to a class in the interface. For example:
   `use Variable as Thing`
2. **use** <*cd-strategy-fragment*> **as** <*interface-class*>
   This maps the targets of a PathSet in the class dictionary to a class in the interface. For example:
   `use (-> *, lhs, Variable) as DThing`

Note that number 1 is a short-hand case for number 2. Number 1 could also be specified as:

```
use (-> *, *, Variable) as Thing
```

which reads as "to Variable through any edge".

Demeter Interfaces solves some of the issues in that it allows us to change the structure without going through and changing all the behavior. It also allows us to specify additional constraints for our strategies which help to verify they are doing and continue to do what we expect them to do.

Demeter Interfaces doesn't solve the issue of splitting monolithic class dictionaries into separate files.

3.3 SML

SML allows you to put types and functions into modules and then use those modules in other modules. You specify which modules are part of your program in the `sources.cm` file [6].

Modules consist of structures and signatures. Signatures specify what's exposed to the outside world and also act as an interface. You write code against the signature and as long as a structure implements the signature, it can be used as an implementation of the signature.

Demeter Interfaces would nicely fit in with SML signatures. The piece that's missing is that DemeterJ (and friends) can't use multiple class dictionaries in a single project. One way to handle this is to build a CM-like system where a DemeterJ project has a project file that specifies all the class dictionaries that belong in the project. Class dictionaries could refer to one another using a module name (perhaps derived from the file name of the class dictionary).

This would allow us to split class dictionaries into multiple pieces, specify which class dictionaries belong to a single project, allow for uniqueness of classes (they're in their individual name spaces), and write behavior and strategies against an interface abstraction of the structure.

DemeterJ (and friends) generate Java code from the class dictionaries, behavior files, and strategies. It's not clear how this generation of classes would work in the presence of a module system where modules are their own name spaces. Would each module become a separate Java package? Would class dictionaries all be in the same package? There are engineering issues with both of these approaches.

3.4 Class graph transformations

Instead of focusing on modularizing class dictionaries starting with how to split up the files, it's useful to think about modularization in terms of transformations on class graphs and class graph composition. The paper Object-extending Class Transformations [8] talks about transforming class graphs to handle evolution of software over the course of the software's life cycle.

Two of the transformations are particularly interesting. We could do composition of class graphs by weekly-extended **GEN** and **ADD**. **ADD** adds new vertices and edges without changing anything else about the graph. **GEN** reroutes construction edges (also known as *is-a* edges) from classes to superclasses.

Aspects are very similar to this. We could use these transformations much like we use aspects by transforming the base class graph to add cross-cutting concerns that are nicely centralized into one place.

Having said that, I think this doesn't fit my intentions. Also, the paper talks about class graphs and as such doesn't get into the issues of name space clashes between graphs or the effects of transforming class graphs on strategies and behavior files.

3.5 Other things to look at

It's probably worth-while to look at LINQ, XML DTDs, SGML (DocBook), and anything else that might have an interesting modularization system.

Additionally, it's probably worth-while to look at formats used by the ontology community like RDF and OWL.

4. CONCLUSIONS

I spent a good portion of the semester trying to work out how to think about modularization of class dictionaries given that class dictionaries are in one sense a textual representation of a class graph and in another sense they're a specification of types that are used in behavior files and strategies. Two thirds of the way into the semester, I switched missions and worked on a paper mapping Demeter concepts to XML Schema and XPath 2.0.

As such I didn't get far enough into the project to get to any conclusions other than that this is a more complex problem than I thought it was when I started. I think if I were to continue on this project, I'd seriously think about working out a similar system for modularization that XML Schema has. I think the two are very similar in the sense that XML Schema is a language for specifying schemas which are hierarchical documents composed of a series of types. Similarly, class dictionaries are textual specifications for class graphs which are composed of objects which in some sense are types.

5. REFERENCES

[1] ANTLR project. http://www.antlr.org/ .

[2] DAJ. http://daj.sourceforge.net/ .

[3] Demeter project. http://www.ccs.neu.edu/research/demeter/ .

[4] DemeterJ. http://www.ccs.neu.edu/research/demeter/software/DemeterJ/ .

[5] DJ. http://www.ccs.neu.edu/research/demeter/DJ/ .

[6] SML/NJ user manual. http://www.smlnj.org/doc/index.html .

[7] D. C. Fallside and P. Walmsley. XML schema part 0: Primer second edition. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/, October 2004.

[8] K. J. Lieberherr, W. L. Hürsch, and C. Xiao. Object-extending class transformations. *Formal Aspects of Computing*, (6):391–416, 1994. Also available as Technical Report NU-CCS-91-8, Northeastern University.

[9] T. Skotiniotis, J. Palm, and K. J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *European Conference on Object-Oriented Programming*, pages 477–500, Nantes, France, 2006. Springer Verlag Lecture Notes.