

Visualization as an Aid to Compositional Software Engineering

Peri Tarr

Harold Ossher

Johannes Henkel

IBM T.J. Watson Research Center

P.O. Box 704

Yorktown Heights, NY 10598

{tarr, ossher}@watson.ibm.com, henkel@cs.colorado.edu

1. Introduction

Software engineering using a compositional paradigm poses somewhat different planning, anticipation and comprehension problems to developers. On the positive side, the use of approaches like multidimensional separation of concerns or aspect-oriented programming promotes development by simplifying the units that developers write. Each unit is devoted to a particular concern, and units tend, therefore, to be smaller than usual and easier to understand. On the negative side, however, there are more units, often many more, and the relationships between them are more complex and more difficult to understand. This complicates the process of composition, or integration, and lessens the developers' understanding of the resulting composed software as a whole.

We have found that, even in seemingly straightforward usages, developers may be unable to predict how the manner in which they separate particular concerns will affect the composition process, and their ability to produce the "right" composed system. As a result, they often produce composed software that does not match either their goals or their expectations, and they have difficulty understanding what was produced, and why it was produced.

In this paper, we describe an early XML-based visualization and navigation tool for Hyper/JTM [7,9] that helps programmers to understand the structure of their concerns and their composed software, and the relationships between them. The goal of this work is, ultimately, to permit software engineers to see and explore, on an ongoing and interactive basis, whether the concerns they are developing and composing together are producing the correct results. When unexpected results are produced, the tool will allow them to explore why this happened and to experiment with alternative separation and composition strategies.

The paper also describes an experiment with the MDSOC paradigm and aspect-oriented development, done in the course of implementing the tool. The generation of XML to power the visualization and navigation was added to Hyper/J as a separately-encapsulated feature, composed using Hyper/J itself. We discovered that some of the simplifications expected from separating these concerns were actually illusory and misleading. With the complexity moved out of the concerns themselves and into the composition/integration stage, we discovered that we had used a design model for the XML feature which, while reasonable and sufficient by itself to implement the new feature, did not take into account some of the complexities of the design model in the core Hyper/J system. As a result, it could not be composed with the core in any way that would produce the desired integrated system.

We suspect that this problem may be endemic to MDSOC and aspect-oriented approaches when used in large-scale software engineering, and, if so, addressing the problem will be critical to the success of this subfield. We describe some of the results of this experiment and suggest how visualization and other techniques may help to ameliorate such problems.

2. Visualizing and Navigating Concerns

Composition-based software engineering raises a number of planning, anticipation, and comprehension issues that do not occur in other paradigms. At the root of these issues is the fact that developers are no longer able to look at a modular chunk of code and understand it in isolation, because concerns or aspects with which it is eventually composed may modify the behavior of that chunk, and so can the particular composition strategy that is used to integrate the pieces. In some ways, these problems are analogous to those that arise in concurrent systems, where developers cannot always tell what interleaving will be produced during any given execution.

There is therefore the need to answer a variety kinds of questions about composed software, such as: What does the composed software contain and what will it do? How does it relate to the inputs? What was a particular input composed into (and what will be affected if I change some part of the inputs)? What did the concerns end up looking like (did I get them right, and did they contain what I expected)? In the case of Hyper/J, what did the entire hyperspace look like? Which parts of it went into a given piece of composed software, and which did not (in case of coverage mistakes)? Why did the compositor produce what it did? Which concern interrelationship(s) caused composition to produce a given entity?

To help developers begin to address these and other critical questions arising from compositional software engineering, we are in the process of developing HyperView, a visualization utility that allows Hyper/J users to browse and explore the results of software composition. Our tool allows one to navigate from a composed entity (such as a hyperslice, class, method, or field) to the corresponding input entities from which it is composed, or navigate from an input entity to the corresponding composed entities. Orthogonal to this composition-based navigation pattern we also provide hierarchical structure: Hypermodules and hyperslices, classes and interfaces, methods and fields and their relationships are represented in our visualization. For all kinds of entities we allow both hierarchical and composition-based exploration.

The implementation is based on XML/web technologies such as XML 1.0 [2], XSLT [3], HTML [4] and ECMA-Script [5] (JavaScript); the visualization output is viewable with any modern off-the-shelf web browser that supports HTML and (optionally) JavaScript. We think this is a good way to introduce a new tool since it minimizes the effort required of a user to try it out. In our opinion, the success of the javadoc-utility [1] can be explained by this observation.

In the JavaScript-enabled version, the screen is divided into a treemenu [6] on the left side and the main area on the right side. As Figure 1 illustrates, the initial screen shows an overview of the hyperspace, in this case one for a small business example involving composition of business rules into a Personnel system. The hypermodules and the hyperslices are listed; the hyperslices are grouped by dimension.

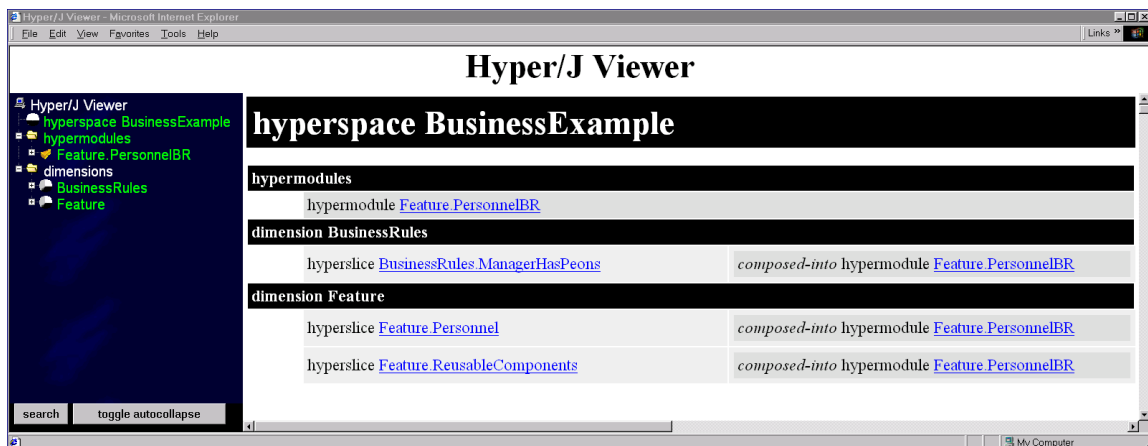


Figure 1. Hyperspace Overview

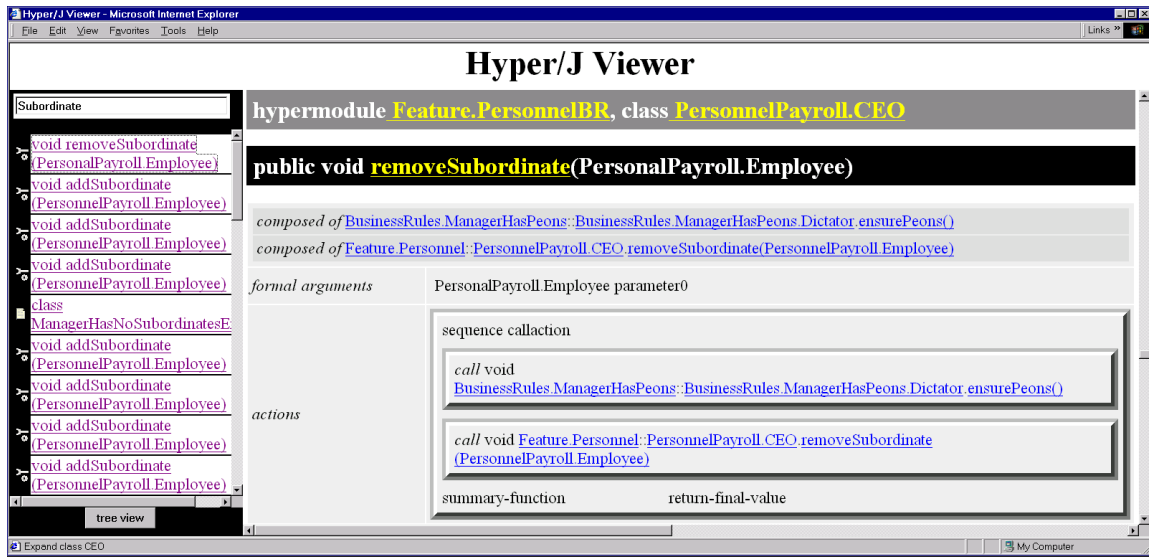


Figure 3. Search Feature

3. HyperView: An Experiment Developing a New Feature Separately

HyperView works off an XML representation of hyperspaces. Hyper/J did not originally generate XML. We decided, therefore, to use Hyper/J itself to encapsulate a new, XML-generation feature as a hyperslice (aspect) and then integrate it with the existing Hyper/J code base using Hyper/J.

Hyper/J uses its own internal model, which we call the *label*, to represent both source and composed Java code. All input hyperslices, as well as the composed hypermodule, are represented by separate labels, with links between them to indicate relationships. Hyperspaces are represented by a separate structure that points into the labels. Thus, we determined that the XML generator feature should produce an XML representation of the relevant parts of labels and the hyperspace representation.

The label is implemented by over 100 Java classes and almost as many interfaces, and the hyperspace representation comprises approximately 20 additional classes and interfaces. Only a small subset, approximately 20-30, of these classes and interfaces need to have distinct behavior with respect to XML generation, however—most of the others can share XML generation behavior and definitions with one of these classes and interfaces. We therefore wrote the XML generation feature as a separate Java package defining just those classes and interfaces having distinct XML generation behavior, with each class containing just those members pertaining to XML generation (plus some abstract methods for declarative completeness). Using Hyper/J, we then treated this new feature package as a hyperslice, and composed it with the existing Hyper/J code base, treated as a second hyperslice. This is the typical manner in which Hyper/J is used to extend an existing system [9].

In the course of doing this, we encountered some problems and observed the potential for some others, even though they did not actually occur in this case. Many of them have been mentioned before by a variety of researchers, but serious work to address them has yet to be done. We believe that such work is critical to the advance of the field of aspect-oriented software development. We discuss some of these problems briefly, with suggestions of how visualization might help.

- **Interference.** Each node of the composed label has a “composedOf” list of all the input nodes used to compose it. For XML generation, we also required the inverse: a “composedInto” list for each input node. As a quick hack to get going, we implemented a method to compute this based on “composedOf” lists, as part of the XML generation feature. In parallel, more efficient support for “composedInto” was added to the Hyper/J base system. When it was complete, we forgot for a while to take out the hacked version in the XML feature, resulting in

execution of both in the composed system. Since both produced the same results, this was not a disaster, except performance-wise, but it is representative of an obvious yet difficult-to-find problem. When concerns are composed, we expect some interaction – that’s why we compose them – but sometimes unexpected and undesired interactions occur.

Visualization can help here by showing what interactions are actually occurring, in a way that allows the user to scan them for unexpected cases. For example, all composed methods that are in fact composed of more than one input method can be shown, analogous to the support in AspectJ environments for showing where in the base code other aspects are woven in [8]. Perhaps this could be summarized by package, by operation (generic function) or in other ways. The intent of this sort of summarization is to reduce the amount information presented to the user, so that the “bad” cases will stand out more easily. We expect that sophisticated program analysis, static and/or dynamic, especially in the face of even partial specifications, could lead to better filters and better summarization; a simple example is to highlight composed methods when both affect the same instance variable as a likely site of interference. This remains an open research area.

- **Unexpected impact on subclasses.** If some new functionality is composed into a class, the question arises as to whether it should also be composed into (or be inherited by) subclasses. It seems natural that it should: in standard programming, if you enhance a class by editing, its subclasses automatically inherit the enhancement, unless they explicitly override it. Hyper/J provides this semantics for composition: an extension class composed with some base class will also be composed with all its subclasses, except where some other extension class is composed (explicitly, or using name correspondence) with the subclass.

The problem we observed was that it was easy to miss subclasses that needed special attention, even some whose very existence we forgot about, which then had inappropriate behavior composed with them. The class hierarchy we used for the XML generation feature was a lot simpler than that of Hyper/J itself. This was deliberate: it was intended to be a good model for XML generation, not for labels and hyperspaces in general. Focusing on the needs and structure of XML generation made it easier to miss issues in the Hyper/J class hierarchy designed with different functionality in mind, and hence not to reconcile the different views appropriately.

This particular problem might seem to be an artifact of the Hyper/J approach of composing separate class hierarchies, each with its own domain model. However, we believe that analogous problems arise with aspect-oriented or compositional approaches in general. When some new functionality has to be composed with a set of classes, there is always the problem of specifying the classes in that set. There are many approaches, including explicit lists, patterns or queries of various kinds, or deductions like use of subclasses. Whatever approach or combination of approaches is used, it is easy in practice to leave out some classes or to include too many. Visualization can help by showing where the functionality is actually included. With the aid of analysis, perhaps as simple as finding all subclasses of a class, visualization can also show classes into which the functionality was not composed, but perhaps should have been. As noted earlier, summarization and filtering are important, aided by analysis. It might also be valuable to by allow the developer to direct the visualization by asking questions, such as “Did this composition affect any classes outside package P,” or “Did this composition affect any classes that are not subclasses of C.” Similar issues and approaches arise with method signature mismatches, where errors can result in multiple overloaded methods instead of a single composed method.

- **Semantic mismatch.** We encountered a fairly simple, but illustrative, case of this in adding the XML feature. The top of the label class hierarchy is a class called NodeImpl. In the XML feature, we added some abstract methods to NodeImpl for XML generation, to be filled in by subclasses, thereby making NodeImpl abstract. In Hyper/J, however, NodeImpl is a concrete class, though some methods are implemented to throw a “not applicable” exception. These two different assumptions clashed, causing the composed NodeImpl and some of its subclasses to be inappropriately abstract.

In general, many assumptions are made in the coding of any concern, and they are usually not made explicit. When concerns are composed, assumptions might clash. It is doubtful whether visualization alone can help with this, but combined with approaches and tools for specifying or deriving assumptions and checking their

consistency, it can be. Prevention will be extremely important, since changing assumptions in an already-coded concern is very costly, so it will be important to move the composition paradigm up earlier in the lifecycle.

- **Unpredictability of composition in general:** It is a well-known problem in concurrent computing that code for different threads might look, and be, correct separately, but parallel execution can lead to race conditions and other bad interactions. After finding that these problems are anything but obvious and result in mysterious behavior at runtime, that community developed a plethora of techniques and tools to avoid and detect such problems. An analogous situation pertains to composition. Compositors integrate concerns in powerful ways, and, though they are deterministic, they are complex enough that the developer might not be able to predict precisely what will happen, and might be surprised at runtime. The aspect-oriented software development community therefore also needs to develop tools and techniques to address this problem.

Visualization helps by showing actually happened, and this is a important start. However, in large systems, developers are still bound to miss things due to the sheer size. In addition to good and flexible filtering in the visualization tool, one needs hooks into a debugger to help diagnose errors that do occur at runtime, and tools involving specification, analysis, checking, verification, and testing in the presence of composition. How to do this in a usable fashion is an open research problem.

4. Conclusions and Future Work

Support for the representation and encapsulation of crosscutting and overlapping concerns is important, because it helps to reduce complexity of individual concerns. But it is only the tip of a large iceberg. The concomitant use of the compositional paradigm, needed to build running systems from the separate concerns, moves complexity from the concerns into the compositor. In many cases, we believe that there is an overall reduction in complexity, because of exploitation of regularities (e.g., weaving a single aspect into multiple classes, or using name matching to integrate very similar class hierarchies). However, irregularities abound in large systems, and so the composition process can be highly complex and can introduce nasty surprises, with detection of resulting errors occurring late in the lifecycle, perhaps as late as runtime. Our own experiences, even as knowledgeable proponents of this paradigm, serve to underline the potential severity of these problems. We believe that they pose a critical challenge in the area of aspect-oriented software development.

Visualization will be an important tool in aiding software engineers to address some of these problems. Simple visualization, such as that currently provided by HyperView, allows developers to explore the composed software and its relationship to the inputs, to check for the presence of expected results and the absence of unexpected ones. This is just a beginning, however. In large systems, even the visualizations will be sufficiently complex that developers will probably miss critical information. Other technologies must be brought to bear also, such as specification, analysis, checking, verification, testing and debugging, well integrated with visualization, to highlight problems or potential problems and help the developer to deal with them. How to do this is an important, exciting, and open, area for future research.

References

- [1] Javadoc Tool Homepage. <http://java.sun.com/j2se/javadoc/index.html>
- [2] Extensible Markup Language (XML) 1.0. <http://www.w3.org/TR/REC-xml>
- [3] XSL Transformations (XSLT) Version 1.0. <http://www.w3.org/TR/xslt>
- [4] HTML 4.01 Specification. <http://www.w3.org/TR/html401/>
- [5] Standard 3rd edition. ECMA-262 <ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf>
- [6] Morton's Javascript treemenu. <http://www.treemenu.com/>
- [7] Hyperspace web site, <http://www.research.ibm.com/hyperspace/>
- [8] AspectJ web site, <http://aspectj.org/>
- [9] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns and the Hyperspace approach." In *Software Architectures and Component Technology: The State of the Art in Research and Practice* (M. Aksit, ed.), Kluwer, 2001 (to appear).