Hyper/J™: Multi-Dimensional Separation of Concerns for Java™

Harold Ossher and Peri Tarr
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
+1 914 784 7975
{ossher, tarr}@watson.ibm.com

ABSTRACT
Hyper/J™ supports flexible, multi-dimensional separation of concerns for Java™ software. This demonstration shows how to use Hyper/J in some important development and evolution scenarios, empahsizing the software engineering benefits it provides.

Keywords
Separation of concerns; multi-dimensional separation of concerns; decomposition; composition; Hyper/J™.

1 INTRODUCTION

Separation of concerns is at the core of software engineering. Done well, it can provide a host of crucial benefits: *additive*, rather than *invasive*, change and low impact of change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse, particularly of off-the-shelf components; simplified component integration; and the ultimate goal of "faster, safer, cheaper, better" software.

To benefit from separation of concerns, one must have the right software modularization at the right time: the concerns that are separated must match the concerns one needs to deal with. Unfortunately, different development activities often involve concerns of dramatically different kinds. For example, changing a data representation in an object-oriented system might involve a single class, or a few closely-related classes, and might be done non-invasively using subclassing or suitable design patterns. Here the hallmark of object orientation—modularization by class (or object)—is a major asset. On the other hand, adding a new feature to a system typically involves invasive changes to many classes, because the feature code is *scattered* across multiple classes, and *tangled* with other code within those classes. Sometimes one needs modularization by class, sometimes by feature, sometimes by other criteria (e.g., "aspect" [5], "role" [1], "variant," etc.), and sometimes by many at the same time.

These considerations led us to identify the need for *multi-dimensional separation of concerns* [9]: the ability to support clean separation of multiple different kinds of potentially overlapping concerns simultaneously, with *on-demand remodularization.* A developer can choose the best modularization, based on any or all of the concerns, for the development task at hand. In addition to reducing impact of change substantially, this opens the door to non-invasive system refactoring and reengineering. Support for on-demand remodularization is a major advance over earlier mechanisms, such as subject-oriented programming [3] and aspect-oriented programming [5], which support more flexible modularization than the object-oriented paradigm they extend, but in only one way at a time.

Our approach to achieving the goals of multi-dimensional separation of concerns is called *hyperspaces* [6, 4], because it involves organizing software in a multi-dimensional space. The dimensions are the kinds of concerns of interest, the points on each dimension are specific concerns, and the location of software units within the space make the concerns they address explicit. Sets of units can be selected, based on the concern structure, to form modules, called *hyperslices,* which encapsulate concerns. Relationships among hyperslices can be specified, and can be used to control flexible *composition* of hyperslices into hypermodules. Sets of hyperslices thus represent different decompositions of the software, and composition allows systems and components to be built using whatever decompositions are desired. Hyperspace technology is language-independent, and can be applied to any programming paradigm, including object-oriented. It augments existing paradigms, which traditionally support a single, dominant means of decomposing systems (by class or object in the object-oriented paradigm; by function in functional languages; etc.).

This demonstration will present Hyper/J™, a tool that supports hyperspaces for Java™. It works with standard Java software, which need not have been developed using Hyper/J or multiple concerns. It requires only standard "class" files as inputs, and produces class files as outputs, so it can operate on binary Java components. It can be used throughout the software lifecycle, such as for initial development, evolution, extension, or integration.

We will demonstrate the use of Hyper/J in several software development and evolution activities, and thereby indicate how multi-dimensional separation of concerns using hyperspaces promotes flexible separation and modularization of concerns, system composition and integration (resulting in desirable properties like mix-and-match), non-invasive evolution and adaptation, and non-invasive, on-demand remodularization. In so doing, it achieves some key software engineering goals, including improved comprehensibility, simplified evolution, and traceability across the software lifecycle; it also greatly simplifies and helps to promote reuse.

2 HYPER/J

Hyper/J supports multi-dimensional separation of concerns for Java using the hyperspace approach. It permits the identification, encapsulation and integration (though composition) of multiple dimensions of concern. It includes a *visual compositor tool*, which provides the ability to identify concerns, including ones that were not identified during initial system development, specify hyperslices in terms of those concerns, and synthesize systems and components by integrating these hyperslices. Hyper/J provides visual, WYSIWYG support for building and editing *composition relationships*, which describe the interrelationships and interactions among concerns in different contexts, and indicate how to build new concerns out of existing ones. Hyper/J can be used at all stages of the software lifecycle, for initial development as well as for extension or evolution of software initially developed with it or without it.

Hyper/J supports the following activities, all illustrated during the demonstration:

- Specification of the set of Java files to consider.
- Specification a *concern mapping* that *identifies* all the concerns that each class and member affects.
- Selection of concerns to be *encapsulated* in hyperslices.
- Specification of *composition relationships* that control the composition of hyperslices into hypermodules.
- Generating composed Java classes.

Hyper/J allows trial-and-error specification of the composition relationships. The user starts this activity by choosing an overall default composition relationship, which is applied to the selected hyperslices to produce a composed hypermodule. Users can then tailor the composition using a variety of commands provided through the GUI. All changes are recorded as composition relationships, which can be viewed, manipulated, and saved. If the input concerns change, the relationships can be reapplied, to yield a result that, in many cases, will be either correct or close to what is desired. Any relationships that are no longer valid will deactivate themselves. The user can interact further, improving the result in the light of the new inputs.

The development of Hyper/J was influenced by some important design goals, intended to foster easy, *incremental* adoption. First, we did not want to require developers to adopt new programming languages, or to use special-purpose compilers or virtual machines. We therefore implemented Hyper/J to work on and generate standard Java class files. All the support for multi-dimensional separation of concerns occurs *outside* the artifact language, Java. Second, we wanted Hyper/J to provide useful benefits when applied to standard Java programs, and additional benefits when applied to programs written with Hyper/J in mind. It is therefore able to identify, encapsulate and integrate concerns from standard Java programs, without requiring special coding conventions or packaging.

A batch version of Hyper/J is available for download, free, from http://www.alphaworks.ibm.com/tech/hyperj.

3 THE DEMONSTRATION

To convey a sense of the different ways in which developers can leverage Hyper/J's capabilities throughout the software development lifecycle, we outline here the scenario we will demonstrate: the development and evolution of a software engineering environment (SEE) that facilitates the development of programs consisting of expressions. The source code and a detailed description of the scenario are part of the Hyper/J release. The (informal) requirements specification for this environment (introduced originally in [9]), are as follows:

> The SEE supports the creation and manipulation of expression programs. It contains a set of tools that share a common representation of expressions. The set of tools should include the following: an *evaluation tool*, which determines the result of evaluating an expression and displays it; a *display tool*, which depicts an expression program textually to a default display device; and a *check tool*, which checks an expression program for syntactic and semantic correctness.

Stage 1: Initial Development, without Hyper/J

To illustrate incremental adoption of Hyper/J, we assume that the initial SEE was developed using standard object-oriented design and implementation techniques, without Hyper/J. Accordingly, a class was designed and implemented to represent each kind of expression. Each class contains constructor, accessor and modifier methods, plus methods eval(), display(), and check(), which realize the required tools in a standard, object-oriented manner. This has the advantage that polymorphism is used effectively: each object knows how to evaluate, display and check itself. However, feature concerns are not identified or encapsulated within this code, despite being a key focus of the requirements specification; instead, the code for each feature (tool) is scattered across all the expression classes.

Stage 2: Mix-and-Match in Retrospect
After using the SEE for a while, the clients indicate that they would like the ability to run different *variants* of the SEE, in which only a subset of the capabilities are present. This is essentially a request to be able to "mix and match" tools in the SEE. Thus, we can think of the SEE as representing a *family* of software [8], where each member of the family contains some combination of tools.

Mix-and-match was not a planned extension. Making the changes to satisfy this rather simple requirement change is no simple matter with standard technology: allowing selection of features requires substantial reengineering, probably to introduce design patterns, like Visitor [2].

We will demonstrate how Hyper/J supports *on-demand remodularization,* in which the feature concerns are identified and encapsulated, in retrospect, and are then composed selectively to form variants of the SEE. This involves:

- Specifying all the Java files that make up the SEE, using a file browser within Hyper/J.
- Specifying a *concern mapping* that introduces the Feature dimension and the feature concerns within it, and indicates which classes, methods and instance variables pertain to which features.
- Selecting desired features and performing composition.
- Generating and executing composed Java classes.

All this will be accomplished without changing, or even recompiling, any of the SEE source code.

This part of the scenario will thus demonstrate the utility of Hyper/J's on-demand remodularization and integration capabilities on *existing, off-the-shelf* Java code. Notice that the feature concerns did not have to be identified or separated during initial development to permit them to be encapsulated "in retrospect." Also, each of the newly-identified feature concerns will itself be a reusable component that can be integrated in different contexts with different other concerns—none of them is coupled with any other. These properties imply powerful support for development and configuration of variations within product lines or families.

Stage 3: Adding a Style Checker
At a later point, SEE clients request an enhancement that permits optional *style checking* of expression programs, in addition to, or instead of, the existing check tool. We will demonstrate how Hyper/J allows the new feature to be developed separately from the existing features, and incorporated non-invasively. This involves:

- Writing the code for the new feature as a new, separate Java package (or packages), and telling Hyper/J to include its Java files in the SEE. We call such a package a *concern package*, or, in this specific case, a *feature package*, because it is deliberately written to encapsulate a feature.
- Specifying a trivial concern mapping that indicates that the whole feature package belongs to a new StyleChecker feature.
- Selecting desired features, possibly but not necessarily including style checking, and performing composition.
- Generating and executing composed Java classes.

This ability to write code as concern packages adds tremendous flexibility to the code architectures that developers can select, and to the range of software development processes they can use.

As we will show, the only code in the feature package is that specifically needed to implement the style checking. Its class structure is similar to that of the original system, but not identical, because style checking only affects some of the Expression classes. This is an important feature of hyperspaces: that different concerns can have different perspectives on, or views of, the domain model under development. These different views can later be reconciled by specifying appropriate relationships between the concerns.

The addition of style checking will thus demonstrate an important feature of Hyper/J: developers need not use Hyper/J during initial development, but if they choose to use it during initial development of some part of the system, they can achieve separation of concerns, and code architectures, that would be difficult or impossible to achieve using standard object-oriented techniques. The extra flexibility does not require the use of new languages or paradigms—the style checker, for example, was written as a standard package in Java—but, instead, is provided by the integration (composition) features of Hyper/J.

Stage 4: Bugging the Code
In the final part of the demonstration scenario, the SEE clients request the ability to log, selectively and optionally, the execution of the SEE. This modification entails making some or all methods in various classes or features print log messages upon method entry and exit. Notice that logging is not the same kind of "feature" as the other SEE tools—it is not a coherent tool itself, and it may (optionally) affect some or all of the features during any execution of the SEE.

Adding support for optional logging, using standard object-oriented mechanisms, would require invasive changes to every method to be logged, such as to perform the logging directly or to participate in Observer design patterns [2].

Clearly, the logging capability is not specific to the expression SEE—it makes no reference to any expression classes or methods, and the same logging capability could be used in multiple contexts. Thus, our demonstration assumes that we already have a library of reusable components that contains an implementation of the Observer design pattern, along with a particular instantiation of that pattern to implement logging. In this case, we demonstrate how to use Hyper/J to retrofit these components, by integrating them into the SEE. This involves:

- Telling Hyper/J to include the Java files for the desired library components in the SEE.
- Specifying a trivial *concern mapping* that specifies that these components belong to a new Logging feature.
- Selecting desired features and performing composition.
- Tailoring the composition by interacting with Hyper/J to specify composition relationships: to which classes and methods should logging be applied?
- Generating and executing composed Java classes.

Hyper/J thus permits us to adapt the library components to this particular use *additively*, without changing either them or the original SEE.

This part of the scenario will thus demonstrate the ability to use Hyper/J to (a) customize and integrate reusable components into a new context, and (b) non-invasively retrofit and integrate design patterns into existing code.

4 CONCLUSIONS

This demonstration will highlight the use of Hyper/J in several common software development and evolution activities, and will show how Hyper/J promotes a number of key activities, including:

- **Flexible separation and modularization of concerns, and non-invasive, on-demand remodularization**: Java software can be written with or without multi-dimensional separation of concerns, and with or without Hyper/J. Hyper/J permits the identification, encapsulation, and manipulation of concerns in standard Java software, either during initial system development or in retrospect, as the need arises during the course of evolution. The tool set works with any Java class files. The scenario demonstrates on-demand remodularization by non-invasively identifying and encapsulating the new feature dimension, and shows how software that was originally written with all features *tangled* together (not modularized) can have those features teased apart and encapsulated as first-class hyperslices without modifying the original software.
- **Composition**: Given a set of hyperslices encapsulating different kinds of concerns, Hyper/J provides the ability to synthesize and integrate some or all of these concerns into systems and system components. It also facilitates mix-and-match and plug-and-play non-invasively, and can aid in the development of product families and product lines.
- **Evolution**: Hyper/J facilitates *additive*, rather than *invasive*, changes for many common evolutionary activities.
- **Adaptation and use of reusable components**: A common problem in software development and evolution is the need or desire to reuse existing components in new contexts—ones for which they will require some degree of specialization or context-specific adaptation. Standard object-oriented (and other) mechanisms are inadequate to permit readily such adaptation and integration without some degree of invasive changes, unless significant pre-planning occurred. Hyper/J can facilitate many forms of non-invasive adaptation and integration of reusable components.

REFERENCES

1. E. P. Andersen and T. Reenskaug. “System Design by Composing Structures of Interacting Objects.” Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1992.
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software.” Addison-Wesley, 1994.
3. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428, September 1993. ACM.
4. Hyperspace web site, http://www.research.ibm.com/-hyperspace.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. “Aspect-Oriented Programming.” In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
6. Harold Ossher and Peri Tarr. “Multi-Dimensional Separation of Concerns and the Hyperspace Approach.” *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000. (To appear.)
7. David L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules.” *Communications of the ACM*, vol. 15, no. 12, December 1972.
8. D. L. Parnas, On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, 2(1), March 1976.
9. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.