IBM Research

Hyper/J™ User and Installation Manual

http://www.research.ibm.com/hyperspace

Peri Tarr
Harold Ossher

Chapter

1

Introduction

What is Hyper/J™?

Hyper/J™ supports advanced, “multi-dimensional" *separation and integration of concerns* in standard Java™ software. This facilitates adaptation, composition, integration, improved modularization, and even non-invasive remodularization of Java software components.

Separation of concerns is simply an approach to decomposing software into modules, each of which deals with, and encapsulates, a particular area of interest, called a *concern*. Examples of concerns are functions, data types or classes, features (e.g., "persistence," "print," or "concurrency control"), variants, and roles. Object-oriented languages permit decomposition by *class*, but only by class. Unlike classes, other kinds of concerns cannot be encapsulated in single modules; instead, their implementations end up *scattered* across the class hierarchy. With Hyper/J, developers can decompose a program according to these other concerns, in addition to classes. They can create new separate modules, in standard Java, that encapsulate these concerns from scratch, without modifying the rest of the program or interfering with the work of other developers, or they can extract such modules from existing Java programs. They can then integrate some or all of these modules to yield programs executable on standard Java virtual machines (JVMs). They can even create multiple system decompositions *simultaneously*, such as by object, by feature, and by product line, and they can add new decompositions at any stage of the software development lifecycle. Hyper/J helps manage the interactions across different decompositions.

Hyper/J provides a powerful *composition* capability, which can be used to combine separated concerns *selectively* into an integrated program or component. For instance, it can be used to create a version of a software system that contains some features, but not others, even if the original system was not written with the features separated. It can be used to extend or adapt a component, even if that component was not written with suitable “hooks,” design patterns [gam94], or open points. For example, suppose a developer needs to produce an XML representation of a complex domain model that spans a large system. Rather than modifying the classes involved in the domain model to add XML streaming methods, (s)he can code these methods in a separate package (or packages), and integrate them with the domain model classes using Hyper/J.

Hyper/J can be used at any stage of the software lifecycle—design, implementation, integration, system evolution and reengineering. When used during the design or implementation of system components, Hyper/J permits developers to architect the system or components to separate *all* concerns of importance from the start. For example, a developer could separate both class and feature concerns, while still using standard Java(TM). When used during system integration, Hyper/J's composition mechanism can be used to integrate separately developed software, including reusable components, and to customize and adapt the software as needed for use in the particular

context. When used during system evolution, Hyper/J's separation of concerns mechanism allows developers to focus on just those pieces of the system that are relevant to the evolutionary path, and its composition and adaptation mechanisms make many forms of evolution possible without changes to existing code. When used during reengineering, Hyper/J's ability to introduce new decompositions without code changes is especially valuable.

Organization of Document

This document describes the installation and use of Hyper/J. Chapter 2 explains how to obtain and install Hyper/J, and how to troubleshoot some common problems that may arise during installation. Chapter 3 introduces software engineering using multi-dimensional separation of concerns and discusses the concepts that underlie Hyper/J. Chapter 4 discusses the use of the Hyper/J tool, describing its command-line syntax, the formats of its inputs and outputs, and the causes of some common error messages. It also explains the current limitations of the tool. Finally, Chapter 5 presents an example software development and evolution scenario and demonstrates how Hyper/J can be used to address common problems in software development and evolution.

In Case of Problems... hyperj-support@watson.ibm.com

While our goal is to make Hyper/J and its accompanying documentation as easy to use and error-free as possible, no software or documentation is completely free of problems. If you run into problems, please send mail to hyperj-support@watson.ibm.com that describes the problem you are having. Although we cannot guarantee that we will be able to fix your problem, we will endeavor to help you as much as we can.

We Want to Hear From You

We are always striving to improve Hyper/J to enable it to address real-world software engineering needs as effectively as possible. We welcome your constructive comments on any aspect of Hyper/J and its documentation. We would also like to hear about any uses that you make of Hyper/J, if you can are able to share them with us. Please send feedback and thoughts to hyperj-support@watson.ibm.com.

Chapter

2

Obtaining and Installing Hyper/J™

2.1. How to Obtain Hyper/J

The Hyper/J binary (plus its supporting documentation) is available without fee on IBM's alphaWorks web site, http://www.alphaworks.ibm.com. Source code is not currently available without special licensing arrangements.

If you are unable to download Hyper/J from alphaWorks, or if you have any problems with the software, please send mail to hyperj-support@watson.ibm.com. We may be able to provide Hyper/J for you on a CD or other medium.

2.2. System Requirements

Hyper/J™ is written in standard Java™, and it is released as a standard jar file. It has no known operating system or virtual machine dependencies. It has been tested with several versions of Sun's JDK, from 1.1.5 to 1.2.1.

The jar file is 1.6Mb in size. Hyper/J should therefore be run on machines with sufficient processing power and memory to handle large Java™ programs.

2.3. Contents of the Release

Hyper/J and its supporting documentation come in a zip file. The zip file contains the following:

- **bin** directory:
  - **hyperj.jar**: A JAR file containing the Hyper/J class files.
- **doc** directory:
  - **hyperj-user-manual.ps**: This file.
- **demo** directory: This directory contains the Java source code and various Hyper/J specification files (described in Chapter 4) for an example software engineering environment application (elaborated in Chapter 5).

2.4. Installing Hyper/J

Once you have downloaded the zip file containing the Hyper/J release from AlphaWorks, you must choose a directory in which Hyper/J will reside. We will refer to this directory as `%HYPERJ_DIR%` (using Microsoft Windows environment variable syntax). Then do the following:

- Move the Hyper/J zip file into `%HYPERJ_DIR%`.
- Unzip the Hyper/J zip file. Unzipping will create the subdirectories noted in Section 2.3. The Hyper/J JAR file, **hyperj.jar**, is in the **bin** subdirectory.
- Add `%HYPERJ_DIR%/bin/hyperj.jar` to your CLASSPATH.

At this point, you can run Hyper/J using the command

```
%JAVA_COMMAND% com.ibm.hyperj.hyperj <command line options>
```

where `%JAVA_COMMAND%` is the command you invoke to run your Java interpreter; for example, the command is `java` when using Sun's JVM. The command-line options are described in Section 4.1.

For your convenience, we recommend defining a batch file or some form of shell script to run Hyper/J. This script can set the CLASSPATH and run Hyper/J.

2.5. Installation Problems

In the unlikely event that you encounter difficulties in downloading or unpacking Hyper/J, please contact hyperj-support@watson.ibm.com. Be sure to include the following information in your message:

- Your operating system.
- The version of Hyper/J you downloaded.
- The error message(s) you are seeing.

Chapter

3

An Introduction to Multi-Dimensional Separation of Concerns

3.1. Separation of Concerns

Separation of concerns [par72] is at the core of software engineering, and all developers do it. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many different *kinds* of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle. For example, the prevalent kind of concern in object-oriented programming is the *class;* each concern of this kind is a data type defined and encapsulated by a class. *Features* [tur98], like printing, persistence, and display capabilities, are also common concerns, as are *aspects* [kic97], like concurrency control and distribution, *roles* [and92], *viewpoints* [nus94], variants, and configurations. We refer to a *kind* of concern, like class or feature, as a *dimension* of concern. Separation of concerns involves decomposition of software according to one or more dimensions of concern. Achieving a "clean" separation of concerns can help

- reduce software complexity and improve comprehensibility.
- promote traceability within and across artifacts and throughout the software lifecycle.
- limit the impact of change, facilitating evolution and non-invasive adaptation and customization.
- facilitate reuse.
- simplify component integration.

These goals, laudable and important as they are, have not yet been achieved in practice. This is primarily because the set of relevant concerns varies over time and is context-sensitive—different development activities, stages of the software lifecycle, developers, and roles often involve concerns of dramatically different kinds and, hence, multiple dimensions. Separation along one dimension of concern may promote some goals and activities, while impeding others; thus, *any* criterion for decomposition and integration will be appropriate for some contexts and requirements, but not for all. For example, the by-class decomposition in object-oriented systems greatly facilitates evolution of data structure details, because they are encapsulated within single (or a few closely related) classes, but it impedes addition or evolution of features, because features typically include methods and instance variables in multiple classes. Further, multiple dimensions of concern may be relevant *simultaneously*, and they may overlap and interact, as features

and classes do. Thus, modularization according to different dimensions of concern is needed for different purposes: sometimes by class, sometimes by feature, sometimes by viewpoint, aspect, role, or other criterion.

3.2. An Example: Expression SEE

To illustrate some of the serious and ubiquitous problems in software engineering that are caused by the tyranny of the dominant decomposition, we begin by describing a running example involving the construction and evolution of a simple software engineering environment (SEE) [tar99]. We will use this example for illustrative purposes throughout this manual. Chapter 5 discusses how Hyper/J can be used to facilitate the development of this example, solving the problems raised here.

The SEE aids in the development of fairly simple programs that consist solely of expressions, such as "A=B+5". Expression programs constructed using the SEE are represented using abstract syntax trees (ASTs), as illustrated in *Figure 1*. This environment has a straightforward and commonly used architecture, also shown in *Figure 1*, in which a collection of tools operates on a shared data structure—the AST. Though the example is, of necessity, small and simple, it is typical of a broad class of real systems that involve multiple tools or applications manipulating wholly or partially shared domain models.

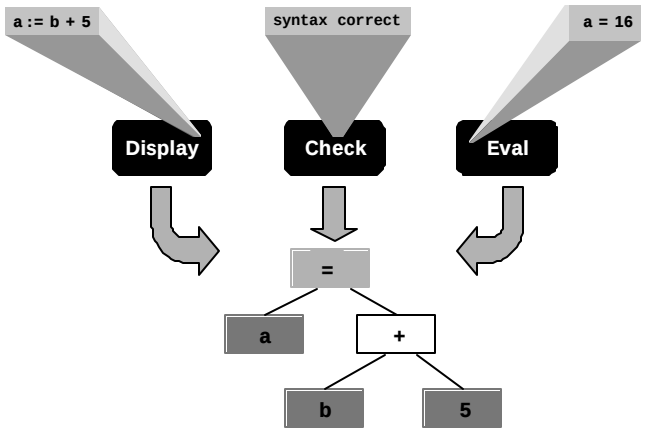

Figure 1. Tools and Shared AST in the Expression SEE.

The running example involves the initial creation of the SEE and a series of evolutionary changes to it. We assume a simplified initial software development process, consisting of informal requirements specification in natural language, design in UML [rum98], and implementation in Java™ [gos96]. The initial requirements specification is straightforward:

> 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:

- **Evaluation tool**: Determines the result of evaluating an expression and displays it.

- **Display tool**: Depicts an expression program textually to a default display device.

- **Check tool**: Checks an expression program for syntactic and semantic correctness.

A straightforward partial UML design for the SEE is shown *Figure 2*. This design uses a standard, object-oriented approach, in which each kind of expression AST node has a corresponding class defined that represents it. 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. The code is structured similarly, and is included in full in the Hyper/J release, in the **demo** directory.

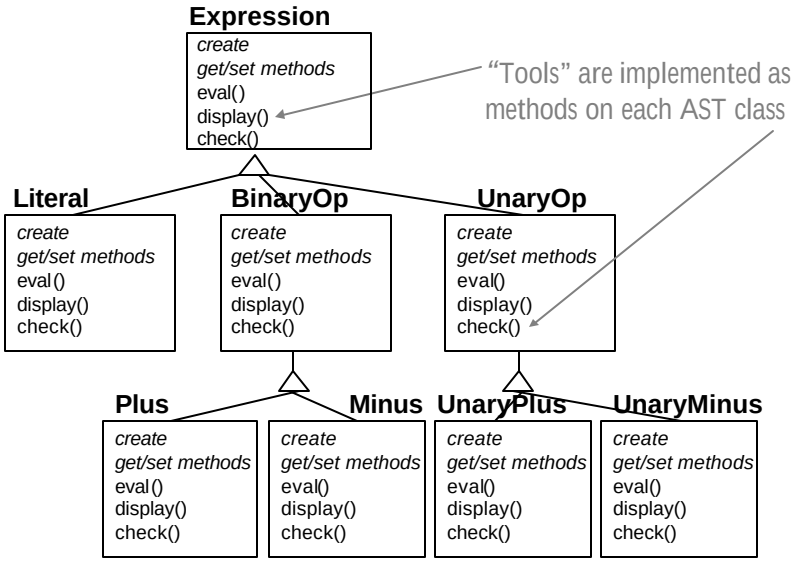

Figure 2. Partial UML Design for the Expression SEE

Even this simple example demonstrates several different kinds (dimensions) of concerns. These include:

- **Classes (or Objects)**: Each of the classes in the design and code represents one class concern.

- **Features**: Particularly from the statement of requirements, we can decompose the software into four coherent features: the "kernel" AST, which includes the actual representation of expressions independently of any of the SEE tools; the display feature; the check feature; and the evaluation feature. Note that each feature includes the corresponding requirement specification, design elements, code, and test cases, since these all pertain to addressing that feature concern in the system.

- **Artifacts**: Traditionally, different stages of the software lifecycle produce different kinds of software artifacts. Some common ones are requirements specifications, designs, code, and test plans.

As noted earlier, we refer to these different kinds of concerns as *dimensions of concern*. Informally, a dimension of concern is simply an approach to decomposing, organizing, and structuring software according to concerns of a particular kind. Note that, despite the clear presence of these different dimensions of concern, only a subset of them can be identified and encapsulated explicitly in the languages used in this example: artifacts, features within the requirements artifact, and objects within the design and code artifacts.

After using the resulting SEE, the clients request some changes:

> It should be possible to have versions of the SEE that include subsets of the tools and capabilities.
>
> It should be possible to impose, optionally, checks for conformance to one or more programming styles.
>
> It should be possible to log, selectively, the execution of the SEE.

This set of modifications suggests the following set of concerns:

- **Configurations**: The first new requirement—to permit different variants of the SEE with different tool configurations—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 [par76], where each member of the family contains some combination of tools.

- **Feature**: Style checking is a new concern in the feature dimension.

- **Logging**: Logging is not the same kind of "feature" as the 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.

- **Design patterns**: While the initial version of the software was simple enough not to require any design patterns [gam94], some of the new requirements present opportunities to benefit from the extra flexibility that design patterns offer. For example, the logging capability could be modeled readily using Observer. From the perspective of comprehensibility, it may be beneficial to look at software in terms of the design patterns from which it is architected [kel99].

Satisfying these rather straightforward requirements is by no means a simple matter with standard object-oriented technology. Allowing selection of features and addition of optional style checking requires substantial reengineering, probably to introduce infrastructure, like design patterns (notably, Visitor), that provides the needed flexibility. Support for logging requires invasive changes to every method to be logged, to perform the logging directly or to participate in Observer design patterns. A more detailed analysis of a similar example appeared in [tar99].

3.3. The Tyranny of the Dominant Decomposition

Even the simple example above illustrates that developers must be able to identify, encapsulate, modularize, and manipulate multiple dimensions of concern simultaneously, and must be able to introduce new concerns and dimensions at any point during the software lifecycle, without suffering the effects of invasive modification and rearchitecture. Modern languages and methodologies, however, suffer from a problem we have termed the *tyranny of the dominant decomposition* [tar99]: they permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are classes (in object-oriented languages), functions (in functional languages), and rules (in rule-based systems). It is, therefore, impossible to encapsulate and manipulate, for example, features in the object-oriented paradigm, or objects in rule-based systems. Thus, it is impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle. Developers of an artifact are forced to commit to one, dominant dimension early in the development of that artifact, and changing this decision can have catastrophic consequences for the existing artifact. What is more, artifact languages often constrain the choice of dominant dimension (e.g., it must be *class* in object-oriented software), and different artifacts, such as requirements and design documents, might therefore be forced to use different decompositions, obscuring the relationships between them.

When software is decomposed into modules based on one, dominant dimension of concern, software addressing other concerns is not localized: it is *scattered* across many modules, and within most of these, it is *tangled* with software addressing other concerns. For example, the “display feature” in the expression SEE is implemented by display() methods in multiple classes. This feature (a concern) is therefore widely scattered across the class hierarchy, and within each class, is tangled with methods for other concerns.

We believe that the tyranny of the dominant decomposition is the single most significant cause of the failure, to date, to achieve many of the expected benefits of separation of concerns.

3.4. Breaking the Tyranny: Multi-Dimensional Separation of Concerns

We use the term *multi-dimensional separation of concerns* to denote separation of concerns involving:

- Multiple, arbitrary dimensions of concern.
- Separation along these dimensions *simultaneously.* No dominant dimension should preclude separation along other dimensions.
- The ability to handle new concerns, and new dimensions of concern, *dynamically,* as they arise throughout the software lifecycle.
- Overlapping and interacting concerns; it is appealing to think of many concerns as independent or “orthogonal,” but they rarely are in practice.

Full support for multi-dimensional separation of concerns opens the door to *on-demand remodularization,* allowing a developer to choose at any time the best modularization, based on any or all of the concerns, for the development task at hand.

Multi-dimensional separation of concerns represents a set of very ambitious goals. They apply irrespective of software development language or paradigm. Our own, evolving approach to satisfying them is called *hyperspaces* (described below). The Hyper/J tool provides hyperspace support for Java™.

3.5. Hyperspaces

Hyperspaces permit the explicit *identification* of any dimensions and concerns of importance, at any stage of the software lifecycle; *encapsulation* of those concerns; identification and management of *relationships* among those concerns; and *integration* of concerns. This section describes hyperspaces in general terms, to introduce the concepts upon which Hyper/J is based.

3.5.1. Concern Space of Units

Software consists of *artifacts*, which comprise descriptive material in suitable languages. A *unit* is a syntactic construct in such a language. A unit might be, for example, a declaration, statement, state chart, class, interface, requirement specification, or any other coherent entity that can be described in a given language. We distinguish *primitive* units, which are treated as atomic, from *compound units*, which group units together. Thus, for example, a method, instance variable, or performance requirement might be treated as a primitive unit, while a class, package, or collaboration diagram might be treated as a compound unit.

A *concern space* encompasses all units in some body of software, such as a set of software systems or component libraries, or a product family. For example, a concern space for the expression SEE contains all of the software artifacts described earlier for both the initial system and the extensions

The job of a concern space is to organize the units in the body of software so as to *separate* all important concerns, to describe various kinds of interrelationships among concerns, and to indicate how software components and systems can be built and integrated from the units that address these concerns. We identify three distinct components to “separation” of concerns:

- *Identification* is the process selecting concerns and populating them with the units that pertain to them.[1] Thus, for example, we can identify the “display feature” concern in the expression SEE as comprising the display requirement and all display() methods in the UML design diagrams and the Java™ code.

- *Encapsulation.* Identification is useful, but to realize fully the benefits of separation of concerns, the concerns must also be *encapsulated* so that they can be manipulated as first-class entities. A Java™ class is an example of an encapsulated concern. The display feature is not an encapsulated concern in Java™, however, as its units are scattered across many Java™ classes.

[1] Note that concern identification can be done either top-down or bottom-up, depending on the stage of the software lifecycle. During design activities, concerns may be selected first, and then units may be developed based on the concerns that were selected. During system evolution, units may already exist when new concerns are identified. In this case, the identification process determines which existing units address the new concerns.

- *Integration.* Once concerns have been encapsulated, it must be possible to *integrate* them to create software that addresses multiple concerns. For example, developers may want to create a version of a software system that contains the "check" and "display" features. In standard Java™, classes are integrated simply by loading them; a combination of import specifications and the class path determines their relationships. Concerns other than classes and interfaces cannot be integrated in standard Java™.

3.5.2. Identification of Concerns: The Concern Matrix

A *hyperspace* is a concern space specially structured to support our approach to multi-dimensional separation of concerns. Its first distinguishing characteristic is that its units are organized in a multi-dimensional matrix. Each axis represents a dimension of concern, and each point on an axis a concern in that dimension. This makes explicit all the dimensions of interest, the concerns that belong to each dimension, and which concerns are affected by which units. The coordinates of a unit indicate all the concerns it affects; the structure clarifies that each unit affects exactly one concern in each dimension.

Each dimension can thus be viewed as a partition of the set of units: one particular software decomposition. Any single concern within some dimension defines a hyperplane that contains all the units affecting that concern. The matrix structure permits uniform treatment of all kinds of concerns, and it allows developers to navigate or slice through the matrix according to any desired concerns.[2]

Some dimensions of concern naturally partition the concern space. For example, if every unit in a system addresses exactly one feature, then the Feature dimension naturally partitions the units. However, some units in a system may not pertain to any "feature" at all, such as an error-reporting routine in the SEE. To handle this situation, each dimension in a hyperspace has a specially-designated "none" concern, containing units that are not of interest from the perspective of that dimension.

3.5.2.1 Units

Hyperspaces can be used to organize and manipulate units written in any language(s), though, of course, tool support is generally language-specific. To date, we have worked only with units at the granularity of declarations (e.g., methods, functions, classes, UML diagrams) rather than lower-level constructs, such as statements or expressions.[3]

Hyper/J treats Java member functions and member variables as primitive units, and interfaces, classes and packages as compound units.

3.5.2.2 Concern Specifications

Concern specifications in hyperspaces serve to identify the dimensions and their concerns, and to specify the coordinates of each unit within the matrix. A simple approach, used in Hyper/J, is a set of *concern mappings* (described further in Section 4.1.1.2) of the form

[2] We believe that the concerns within a dimension, though disjoint, need not be unrelated, and we expect some concern structure (e.g., hierarchies) within dimensions to be valuable [oss88, kim99]. This remains an issue for future research, and is not yet supported by Hyper/J.

[3] We believe that hyperspaces can be extended to handle finer-grained units in a disciplined fashion; this remains an issue for future research, and is not yet supported by Hyper/J.

```
x: dimension.concern
```

where *x* is the name of a unit or a collection of units (e.g., a directory or package), or a pattern representing many units or collections of units.[4]

3.5.3. Encapsulation of Concerns: Hyperslices

The concern matrix identifies concerns and organizes units according to dimensions and concerns. It allows many useful sets of units to be identified based on the concerns they affect, such as all units pertaining to a single concern, or to all of several concerns (areas of overlap), or to one concern but not another. However, the matrix does not, in itself, support encapsulation of concerns: the sets of units cannot simply be treated as modules without additional mechanism. In hyperspaces, that additional mechanism is the *hyperslice*: a set of concerns that is *declaratively complete*, which means that they must *declare* everything to which they refer.

Units are typically related in a variety of ways; for example, one function unit may *invoke* another, or it may *define* or *use* a variable declaration unit. When these kinds of interrelationships exist between units in different concerns, high coupling results. The declarative completeness property of hyperslices is intended to decouple hyperslices from each other. Declarative completeness means, for example, that a hyperslice must, at minimum, include a declaration for every function that any of its members invokes, and for any variable its members use. The hyperslice need not provide a full *definition* for these declarations—e.g., it may declare a function without providing an implementation. Thus, declarations can be abstract, specifying (partially or fully, formally or informally) the properties upon which the hyperslice relies.

Declarative completeness is important because it eliminates coupling between hyperslices. Instead of one hyperslice referring to another, thereby depending upon the other specific hyperslice, each hyperslice states what it needs by means of the abstract declarations, thereby remaining self-contained. It does, however, require someone to provide full definitions of the abstractly-declared entities to be fully complete, but *any* appropriate hyperslice(s) can provide these, through integration. This approach therefore fosters flexible configuration and reuse of hyperslices, and is crucial to achieving limited impact of change.

For example, suppose a Display hyperslice contains a unit, Plus.display(), which uses a Plus.getOperand() accessor function, defined in a Kernel hyperslice. To make Display declaratively complete, it must be augmented with its own declaration of Plus.getOperand() (without necessarily implementing it). Plus.display() must then refer to this local declaration, instead of to the accessor function in the Kernel. This eliminates the coupling between Display and Kernel, in favor of the assertion that the new, abstract declaration must eventually be "bound" to a unit in *some* hyperslice that provides a suitable implementation.

Any set of units can be fashioned into a valid hyperslice by *declaration completion*: providing abstract declarations for everything referenced but not declared within the set.

[4] In general, concern specifications can be more complex, and can specify the "meaning" of each dimension and concern formally or informally. There are two styles of specification. *Extensional specifications* explicitly enumerate the units in each concern. *Intensional specifications* specify properties of concerns and units that can be used to determine whether a given unit pertains to a concern. Intensional specifications have the advantage of conveying intent more explicitly, and of being able to accommodate changes to the underlying set of units without manual intervention.

This process can be performed automatically, using straightforward (though language-specific) analysis; this is done in Hyper/J.[5]

Since any set of units can become a hyperslice through declaration completion, arbitrary concerns can be encapsulated using hyperslices. Thus, whatever limitations the underlying artifact language(s) has, and whatever the concern, it is always possible to synthesize a hyperslice that contains just those units pertaining to the concern (plus some abstract declarations).

3.5.4. Relationships among Concerns

Units, concerns and hyperslices do not exist in isolation; they can be interrelated in a number of different ways. For example, the "display feature" and the "expression class" are related in that they *overlap*—they share some of the same units, as the display() method is part of both concerns—so a change to one concern may affect the other. As another example, we might choose to integrate "syntax check" and "style check" hyperslices to produce a "check" feature that performs both syntax and style checks. In this case, these two hyperslices would be related by one or more *integration* relationships that indicate how they are to be combined.

We can identify two distinct classes of relationships: *context-insensitive* and *context-sensitive*. "Overlap" is an example of a context-insensitive relationship—the "display feature" and "expression class" are always related this way, as long as they share units in common. Integration relationships exemplify context-sensitive relationships—the "syntax check" and "style check" concerns only have this relationship if they are being integrated in some context (e.g., to create a check tool), but the relationship is not inherent in their definition. Other common kinds of concern relationships are "generalizes," "subsumes," and "precludes." Hyperspaces permit the identification and representation of both context-insensitive and context-sensitive relationships, and their use in analysis (e.g., impact of change) and integration, though the current release of Hyper/J supports just integration relationships.

3.5.5. Integration of Concerns: Hypermodules

Hyperslices are building blocks; they can be integrated to form larger building blocks and, eventually, complete systems. For example, to create a working SEE containing the Display hyperslice discussed above, Display must be integrated with some other hyperslice that provides a unit that can be bound to the new, abstract declaration of Plus.getOperand(), to provide an implementation. We refer to this kind of "binding" relationship between units as *correspondence*. Correspondence is a context-sensitive relationship. It occurs within the context of the integration of a particular software component or system—the same declaration unit may be associated, for example, with different implementation units in different systems. In a hyperspace, this integration context is a *hypermodule.*

A *hypermodule* comprises a set of hyperslices being integrated and a set of *integration relationships*, which specify how the hyperslices relate to one another, and how they should be integrated. Correspondence is an important integration relationship, indicating

[5] Automatic declaration completion determines what declarations are needed, and can create valid declarations. Semantic information associated with declarations—formal or informal specifications—is another matter however, and probably requires human intervention. Specifications on declarations, and the extent to which they can be determined automatically by analysis during declaration completion, remain issues for future research, and are not currently supported by Hyper/J.

which specific units within the different hyperslices are to be integrated with one another. However, additional details are often needed to specify just how the integration is to occur. For example, if two methods correspond, should one override the other in the integrated system, or are they both to be executed? If both, in what order, and how should the return value be computed? If the types of their parameters do not match, what transformations are needed to reconcile them? In the example above, it is sufficient to integrate the corresponding declarations of Plus.getOperand() in Display and Kernel, which results in the Kernel implementation being called by Plus.display() at run time. Integration relationships in Hyper/J™ extend the *composition rules* from our earlier work on subject-oriented programming [oss96].

Conceptually, and often in practice through use of a *compositor* tool (such as that included within Hyper/J), the integration specified by integration relationships can actually be performed to produce a set of integrated units. This set will be declaratively complete, and is therefore a hyperslice. A hypermodule can therefore be thought of as a composite hyperslice, produced by integrating a number of subsidiary hyperslices. This implies that hypermodules can be nested, allowing large systems to be built by successive integration.

Declarative completeness, correspondence, and even the more detailed integration relationships, represent fairly loose forms of binding, which promotes evolvability. Since hyperslices do not depend on each other directly, software artifacts are subject to a *completeness constraint* in which each declaration unit in a system must correspond to compatible definition(s) or implementation(s) in some hyperslice(s). Replacing a definition or implementation is non-invasive on hyperslices; it merely requires the redefinition of integration relationships. Correspondence thus provides great flexibility and directly supports substitutability, including mix-and-match and plug-and-play. Completeness constraints can be imposed as needed (e.g., on code, to ensure that it can run), but they are not necessary when a hypermodule represents a building block (e.g., a reusable component or framework), whose remaining needs can be satisfied through future integration.

Clearly, the issue of whether corresponding units are “compatible” (e.g., whether an implementation unit satisfies a declaration unit’s requirements, or whether a design unit satisfies a requirement) involves both syntactic and semantic issues. How to characterize and check for such compatibility remains an issue for future research. Even once resolved, however, we expect checking to be semi-automatic in general; ultimately, software engineers must understand enough about corresponding units to determine whether or not they are compatible and how best to integrate them. Hyper/J currently performs checks for syntactic compatibility only; semantic compatibility is the responsibility of the developer.

Hypermodules can be used to encapsulate many kinds of software artifacts, components, and fragments thereof, and to integrate them in different ways. For example, an entire artifact, like a requirements specification, a design, or code, can be modeled as a hypermodule. A software system as a whole is also a hypermodule, subject to the declarative completeness constraint. A system hypermodule might consist of a hyperslice for each artifact, with correspondence relationships describing how the artifacts interrelate; they might, for example, indicate how particular design and code units elaborate given requirements units. Alternatively, it might consist of a subsidiary hypermodule for each feature, with integration relationships specifying how the features interact. Each feature hypermodule, in turn, consists of a hyperslice for each artifact, with integration relationships as above.

3.6. Hyper/J™

Hyper/J is a tool that realizes hyperspaces for Java™ code. In so doing, it permits developers to achieve improved modularity initially and throughout the course of the software development lifecycle; adaptation, customization, and integration through composition; traceability through correspondence and other relationships across hyperslices; loose coupling and “mix and match” through declarative completeness; and on-demand remodularization.

Hyper/J currently supports units that are Java™ packages, interfaces, classes and members. It supports a concern matrix of these units, and the ability to make hyperslices from sets of units and then to integrate the hyperslices into hypermodules. It generates Java™ class files for all hypermodules produced. These can be executed, if complete, or used as building blocks for further development.

Chapter
4

Using Hyper/J

4.1. Running the Tool

Hyper/J is a standard Java™ application. It is therefore run the same way as any other Java main program—by invoking whatever Java virtual machine you use. For example, if using Sun's JVM, Hyper/J is invoked as:

```
java com.ibm.hyperj.hyperj <options>
```

The fully qualified name (with the whole file path) of the Hyper/J jar file either must be in your CLASSPATH or must be passed as an option to the JVM. For example, if you installed into a directory %HYPERJ_DIR%:

```
java –CLASSPATH=%HYPERJ_DIR%\bin\hyperj.jar com.ibm.hyperj.hyperj
<options>
```

We recommend defining a shell script (Unix), batch file (Windows), or other script to run Hyper/J, to eliminate the extra typing (and possibility of error) that running the tool manually entails.

If you are using a version of Sun JDK *earlier than* version 1.2, you must also include explicitly all of the Java standard libraries in the CLASSPATH.

If you are using Sun JDK 1.2 or higher, you may need to add the following JDK files to your CLASSPATH (if they are not already there):

```
%JAVA_DIR%\lib\tools.jar
%JAVA_DIR%\jre\lib\rt.jar
%JAVA_DIR%\jre\lib\jaws.jar
%JAVA_DIR%\jre\lib\i18n.jar
```

Add these files to your CLASSPATH if you run Hyper/J and it crashes with the exception `java.lang.NoSuchMethodError`; this is usually the cause of this problem.

4.1.1. Hyper/J Required Command Line Parameters

Hyper/J's command-line parameters can be specified in any order. They are currently case-sensitive, however, so please be sure to specify them correctly.

4.1.1.1 Hyperspace Specification File:

The hyperspace specification file (described in Section 4.2.1) is similar to a project description. It lists all of the Java class files with which a developer is working, and to

which the developer wishes to apply Hyper/J. Hyperspace specification files are specified as:

```
-hyperspace c:\users\smith\someProject\project.hs
```

The name of the hyperspace specification file may include either a relative path or a full path.

In some cases, it is possible to omit the –hyperspace specification and to allow Hyper/J to derive one automatically. See Section 4.2.1.3 for a description of this feature.

4.1.1.2 Concern Mapping File(s):

Concern mapping files describe how various pieces of Java class files address different concerns in a hyperspace. Developers must specify at least one concern mapping file when running Hyper/J, but they may specify multiple concern mappings (separated by one or more blank spaces). For example:

```
-concerns concernFile1.cm c:\users\smith\someProject\concernFile2.cm
```

Concern mapping files can be specified using either absolute or relative paths, as shown above, where the file `concernFile1.cm` is taken from the current directory, while concernFile2.cm is taken from the directory `c:\users\smith\someProject`.

The contents of concern mapping files are described in Section 4.2.2.

4.1.1.3 Hypermodules or Relationships Specification File:

The hypermodule specification file (discussed in Section 4.2.3) describes one or more hypermodules—integrations of concerns—that a developer wishes to create. Developers must indicate a hypermodule specification file as follows:

```
-hypermodules ..\someProject\myHypermodules.hm
```

The hypermodule specification file may be specified using either absolute or relative paths.

In cases where developers are working with small projects, they may wish to specify only the integration relationships that are part of a hypermodule specification (see Section 4.2.3), rather than a complete hypermodule specification. In such cases, the developer may use the –relationships option *in place of* the –hypermodules option, specifying instead a file that contains just the appropriate integration relationships:

```
-relationships ..\someProject\myRelationships.hm
```

Section 4.2.3 describes this feature in more detail.

4.1.1.4 Using a Single Control File:

Some users will find that keeping the hyperspace, concern mapping, and hypermodule specifications in separate files (as shown in Sections 4.1.1.1-4.1.1.3) is most convenient for them. Others, who have short specifications or commonly used specifications, may find it more convenient to specify all of this information in a single file, which can be passed as a parameter to Hyper/J. Thus, rather than specifying the -hyperspace, -

concerns and –hypermodules (and other options) separately on the command line and putting these specifications into three separate files, it is also possible to put all of the options into a single file, and to name that file as the first parameter to Hyper/J:

```
java com.ibm.hyperj.hyperj demo.opt –verbose
```

This option indicates that the file `demo.opt` contains all of the option information, including the hyperspace, hypermodule, and concern mapping specifications. The format of `demo.opt` might be:

```
-hyperspace
   <include either a hyperspace declaration or a hyperspace
   specification file name here; see Section 4.2.1 for a description
   of hyperspace specifications>
-concerns
   <include concern mappings, or concern mapping file names, here;
    see Section 4.2.2 for a description of concern mappings>
-hypermodules
   <include hypermodule specification, or hypermodule specification
   file name, here; see Section 4.2.3 for a description of
   hypermodule specifications>
```

This file could also contain any of the other Hyper/J options, such as –verbose.

4.1.2. Optional Hyper/J Command-Line Parameters

Hyper/J supports some additional, optional parameters, which are used to control the tool's output.

4.1.2.1 Output Directory

By default, Hyper/J will place composed class files into a subdirectory of the current directory. (This directory will always have the same name as the hypermodule from which the composed classes were created; if integration relationships are specified instead of a complete hypermodule (see Section 4.1.1.3), the directory name used is Composition.) The `–output` option permits the developer to select a different directory into which to place the subdirectory:

```
-output c:\users\smith\someProject\hypermodules
```

The above example will cause Hyper/J to put the composed class files into a subdirectory of the directory `c:\users\smith\someProject\hypermodules`, rather than into the directory from which Hyper/J was run.

Note that the `–output` option will NOT change the *name* of the subdirectory that Hyper/J creates—it will simply change the *location* of that subdirectory. To change the name of the subdirectory, it is necessary to change the name of the hypermodule in the hypermodules specification file.

The output directory can be specified using either absolute or relative paths. The named directory *must* exist; Hyper/J will not create it if it does not exist.

4.1.2.2 Verbose

The `-verbose` option is used to instruct Hyper/J to print some helpful status information at important points during its execution. It also causes Hyper/J to create *unparse files* (see Section 4.2.5), and a dump of the hyperspace showing the dimensions and concerns that were created and the units that address each concern. These files can be used to help debug erroneous hypermodule specifications (or Hyper/J itself).

4.1.2.3 Debug

The `-debug` option tells Hyper/J to generate various forms of information that are useful for debugging erroneous hypermodule specifications, or Hyper/J itself. This option is recommended for advanced users only, as the information it produces is not readily comprehensible without more detailed understanding of Hyper/J's implementation.

4.2. Formats of Hyper/J Inputs and Outputs

4.2.1. Hyperspace Specification File

Hyperspace specification files are similar to project definitions; they simply describe the set of Java class files with which a developer is working. The format of a hyperspace specification file is:

```
hyperspace hyperspaceName
   classFileSpecification;
   classFileSpecification;
   ...
```

4.2.1.1 Hyperspace Name:

Each hyperspace specification must give a name to the hyperspace. This name should be chosen to be mnemonic—for example, it might be the name of the project (e.g., someProject), or a description of the goals of the hyperspace (e.g., reusableComponentsHyperspace).

4.2.1.2 Class File Specification:

The class file specifications name one or more Java class files that are to be treated as part of the hyperspace. (Note that a given Java class file can be part of multiple hyperspaces.) The class file specifications take one of the following forms (shown by example):

```
class package1.className1, package1.className2;
composable class package2.* except package2.someClass;
composable class package3.* including subclasses;
composable file c:\users\smith\someProject\*;
uncomposable class java.lang.*, java.io.*;
```

Class files can be specified either by using Java fully qualified class (or interface) names (as in the first two lines above), or by using file names (the second two lines). For example, the standard Java utility class Hashtable could be included in a hyperspace by

referring to it by its fully qualified name (`java.util.Hashtable`) or by its class file name (e.g., `c:\programs\java\src\java\util\Hashtable.class`). If the class file name form is used instead of the qualified name, you must include the fully specified absolute path; relative paths are not accepted. If you use the qualified name, you must be sure that your CLASSPATH environment variable is set appropriately, just as you would do for the Java compiler, so that Hyper/J can find the packages and classes to which you refer. Class file specifications do not distinguish classes from interfaces; you may include either in a **class** or **file** specification.

Class file specifications may include simple wildcarding, as shown above. The ***** wildcard indicates that all of the classes or class files in a given package or directory are to be included in the hyperspace. No other wildcards are supported at present. If a developer wants to include *most* of the classes or files in a package or directory, but wishes to exclude a small number of them, s/he can use the **except** clause, as shown in the second line of the above example. That class file specification indicates that all classes in a package named `package2` are to be included in the hyperspace, except for the class `package2.someClass`. Any number of exceptions can be listed, separated by commas.

Class file specifications include the designation **composable** or **uncomposable**; if neither is specified (as in the first line of the example above), the default is **composable**. Composable means that the named classes can be composed with other classes: they might be composed automatically, e.g., with like-named classes in other packages, and they can participate in integration relationships. Uncomposable means that they may not be composed: no automatic composition will occur, and they may not participate in integration relationships. In general, library classes (like the Java predefined classes, such as Object and String) should be treated as uncomposable. Please be aware that if you do not include a class file specification for some class to which one of the included composable classes refers, the omitted class will be included in the hyperspace automatically, but it will be treated as *uncomposable*. This may lead to some composition behavior that you did not expect. For this reason, we recommend listing all of the classes you care about explicitly. If you see unexpected behavior and want to determine whether it is because some classes were treated as uncomposable when you intended to have them be composable, check the unparse files (Section 4.2.5). These indicate, for each class in the hyperspace, whether or not that class is composable.

Whereas classes used by composable classes are included automatically, classes that use composable classes cannot be. *It is critical that all such classes of interest in the project be included in the class file specifications.* If they are not, they will not call the composed classes produced by Hyper/J when they execute, and will therefore not be affected by the composition.

In some cases, users may discover that they would like to make composable any subclasses of a given composable class, without having to list the subclasses explicitly. The modifier **including subclasses** can be used in a **composable class** specification to indicate that any subclasses of the named class should themselves be composable. Hyper/J will search the class path to find the subclasses. By default, these subclasses will be placed in the same concern as their superclass.

4.2.1.3 Omitting the Hyperspace Specification File: A Simplification

In some cases, it is possible to omit the hyperspace specification and let Hyper/J create one automatically. If you do not define a hyperspace specification, Hyper/J will derive one automatically from the concern mapping (Section 4.2.2). It uses the following rules to create the hyperspace specification:

- For concern mappings of the form

  `package P : Dimension.Concern`

  Hyper/J adds each class defined in package P to the hyperspace as a composable class.

- For concern mappings of the form

  `package P as in package Q : Dimension.Concern`

  For every class named C that appears in both packages P and Q, Hyper/J will add P.C to the hyperspace as a composable class.

  Note: This is *not* done recursively! If recursive inclusion is required, the developer must define a hyperspace specification and cannot use the default behavior.

The hyperspace specification that Hyper/J generates is written to a file named `__default.hs`. If the default hyperspace specification is not quite correct, users are free to edit this file and then specify it *explicitly* when running Hyper/J subsequently (using the –hyperspace option).

Notes:

- A current limitation of Hyper/J is that it will only derive hyperspace specifications automatically based on *package* concern mappings. It does not yet treat *class* concern mappings (e.g., “class P.C : Dimension.Concern”).

4.2.2. Concern Mapping File(s)

Concern mapping files define a set of dimensions and concerns that you have, and they describe how the classes and interfaces in your hyperspace, and their members, address those dimensions and concerns (see Chapter 3).

Hyper/J automatically creates one dimension, the C*lass File* dimension, when it processes your hyperspace specification file, and it creates one concern in that dimension for each of the class files it reads as part of the hyperspace (whether you specify them as classes or as class files). The class file dimension is, therefore, the one that you use when you want

to work with standard, object-oriented concerns (classes and interfaces). The concern mapping file is used to describe any other dimensions and concerns that you may have.

Concern mapping files consist of any number of mappings, each of the form:

packageMapping | *classMapping* | *interfaceMapping* | *operationMapping* | *fieldMapping*

We describe the kinds of mappings below.

Notes:

- The processing of concern mapping files follows the general principle that later mappings supercede earlier ones. This permits developers to specify mappings that apply in the general case, and then later indicate exceptions to the earlier mappings. Note, however, that this induces order sensitivity—you often will not see the same results if you write your concern mappings in different orders. If you are not sure about what concern structure your concern mappings have produced, you should use the `-verbose` option to Hyper/J, as this will produce a dump of the hyperspace that you can examine.

- Concern mappings are case-sensitive. Please be sure to check the spelling and capitalization of *all* identifiers, or you will likely see some unexpected results.

4.2.2.1 Package Mapping:

Package mappings are a shorthand way of allowing developers to indicate that the entire contents of a given Java package address one particular concern. For example,

```
package java.lang : Feature.Library;
package someProject.util : Feature.Utilities;
```

The first concern mapping indicates that all of the classes and interfaces, and all of their members, that are part of the package `java.lang` address the `Library` concern in the `Feature` dimension. This mapping has two effects. First, if the hyperspace does not contain a `Feature` dimension or a `Library` concern within that dimension, Hyper/J will create them. Second, it indicates that the contents of package `java.lang` address the `Feature.Library` concern.

Notes:

- If an earlier concern mapping had placed any of the classes or interfaces (or any of their members) defined in package `java.lang` into a different concern in the `Feature` dimension, this later concern mapping would supercede it, and the affected units would be moved to the `Library` concern in the `Feature` dimension. Concern mappings pertaining to other dimensions (such as the class file dimension) are not affected.

4.2.2.2 Class and Interface Mapping:

Class and interface mappings permit developers to indicate that a given class or interface, and all of its members, address a given concern in some dimension. For example,

```
class someProject.util.MyClass : Optimizations.UtilityOptimizations;
interface java.util.Enumeration : OtherDimension.SomeConcern;

class FooClass : Feature.FooClassConcern;
```

The first and second concern mappings above indicate that the Java class `someProject.util.MyClass` (and all of its members) address the `UtilityOptimizations` concern in the `Optimizations` dimension, and that the interface `java.util.Enumeration` addresses the `SomeConcern` concern in the `OtherDimension` dimension. Again, if no concerns or dimensions by these names exist, Hyper/J will create them. The third mapping is a shorthand way to indicate that *all* classes named `FooClass`—no matter which Java package they are in—address the concern named `FooClassConcern` in dimension `Feature`.

Notes:

- If an earlier concern mapping had placed any of the members (methods or member variables) of class `somePackage.util.MyClass` or interface `java.util.Enumeration` into a different concern in the `Optimizations` or `OtherDimension` dimensions (respectively), the later concern mappings would supercede the earlier ones, and the affected members would be moved to the `UtilityOptimizations` and `SomeConcern` concerns in the `Optimizations` and `OtherDimension` dimensions (respectively). Concern mappings pertaining to other dimensions (such as the class file or `Feature` dimensions) are not affected.

4.2.2.3 Operation Mapping:

Operation mappings indicate that one or more operations address a given concern in some dimension. There are two forms of operation mappings:

```
operation SomePackage.SomeClass.someMethod : Feature.SomeConcern;
operation foo : Feature.Foo;
```

The first form indicates that all methods named `someMethod` in class `SomePackage.SomeClass`, irrespective of signature, address the concern `SomeConcern` in the `Feature` dimension. The second form is a shorthand notation for indicating that *all* methods named `foo` in the hyperspace—no matter what class or interface they belong to, or whatever their parameters are—address the `Feature.Foo` concern.

Notes:

- Constructors and static initializers can be mapped to concerns. Their names are `<init>` and `<clinit>`, respectively, as they appear in class files (as defined by the Java language specification).

- Hyper/J is currently limited to specifying operation mappings using only the operation *name*, rather than also being able to include the *signature*. Signatures will be included in an upcoming release.

- As in the other cases, later mappings override earlier ones for the same dimension.

4.2.2.4 Field Mapping:

Field mappings are used to indicate that one or more instance or class (static) variables address a particular concern. There are two forms of field mappings:

```
field SomePackage.SomeClass.instanceVar : Feature.SomeConcern;
field fooVar : Feature.Foo;
```

The first form indicates that the instance variable named `instanceVar` in class `SomePackage.SomeClass` addresses the concern `SomeConcern` in the `Feature` dimension. The second form is a shorthand notation; it indicates that *all* member variables named `fooVar` that occur in the hyperspace—no matter what class they belong to—address the `Feature.Foo` concern.

Notes:

- As in the other cases, later mappings override earlier ones for the same dimension.

4.2.2.5 "None" Concerns:

The fact that later mappings always override earlier ones for the same dimension ensures that a unit can be in at most one concern within any particular dimension. In fact, the hyperspace model requires that it be in *exactly* one concern. Each dimension therefore has a special concern called `None`, and Hyper/J automatically puts any units not assigned to any other concern in the dimension into its `None` concern. None concerns can also be referred to explicitly.

None concerns are useful, because they are an intuitive and convenient place for units that really do not affect a particular dimension. An example is given in Chapter 5.

4.2.3. Hypermodule or Relationships Specification File

A hypermodule specification file is used to define a hypermodule that is a particular integration of the units pertaining to some selection of the concerns in the hyperspace. Following the hyperspace model described in Chapter 3, it identifies some hyperslices that are to be integrated, in terms of the concerns in the hyperspace, and specifies integration relationships that give details of the desired integration:

```
hypermodule hypermoduleName
   hyperslices:
      dimensionName1.concernName1,
      dimensionName2.concernName2,
      ...
   relationships:
      mergeByName | nonCorrespondingMerge | overrideByName;
      other relationships
end hypermodule;
```

In cases where developers are working with small projects, they may wish to specify only the integration relationships, rather than a complete hypermodule specification. In such cases, the developer may define a relationship specification instead of a full hypermodule specification. The relationship specification simply lists the desired integration relationships:

```
mergeByName | nonCorrespondingMerge | overrideByName;
other relationships
```

When a developer uses a relationship specification instead of a full hypermodule specification, Hyper/J will derive the *hypermoduleName* and **hyperslices** (see Section 4.2.3.1 for a description of hyperslices) parts of the hypermodule specification automatically, using the following rules:

- **Hypermodule name**: The name of the hypermodule is always Composition.
- **Hyperslices**: Hyper/J includes as hyperslices all of the concerns that were mentioned in the concern mappings (Section 4.2.2), in the order in which they were mentioned. It does *not* include any of the None concerns, and it does not include any of the concerns in the ClassFile dimension, which are created automatically by Hyper/J.

If these rules will not produce the desired behavior, the developer must specify a complete hypermodule specification, and should *not* use the simplified relationship specification.

Notes:

- The identifiers (hypermodule name, dimension names, and concern names) in hypermodule specifications are all case-sensitive, but the relationship specifications are not; thus, for example, **mergeByName** is equivalent to **mergebyname**.

4.2.3.1 Hyperslices—Concerns to be Integrated:

As described in Chapter 3, hyperslices are sets of units that are to be integrated. Currently, they are specified simply as the names of concerns in the hyperspace. We intend that these specifications will evolve, in future releases, to allow set operations, such as union, intersection and set difference, on concerns.

4.2.3.2 Integration Relationships:

Integration relationships describe how the hyperslices are related and how they are to be integrated together to form a new piece of software that contains some or all of the functionality of the original hyperslices. They do this by indicating which units in corresponding hyperslices match each other and how they should be synthesized together into a new unit.

Note: Some of the relationships described in this section, or some of their options, are not supported by the current release of Hyper/J. Please refer to Section 4.5 for a list of current known limitations.

Some Terminology:

Hyper/J distinguishes between *operations* and *actions*. Both pertain to methods, but we avoid use of the term “method” in an attempt to avoid confusion.

- *Operations* are like generic functions in CLOS or selectors in Smalltalk: they represent a method name and signature, but do not commit to any particular declaration or implementation in any particular class. Operations are typically implemented by multiple classes, just like a method declared within an interface in Java™.

- *Actions,* on the other hand, are implementations of operations for specific classes: actual functions. When we do use the term “method” in the narrative, it is synonymous with action.

There are some contexts in which an action must be specified. An operation specification is usually permitted is such contexts, as shorthand for *all* the actions implementing that operation.

4.2.3.3 Composition Strategy:

The specification of integration relationships in Hyper/J follows an approach where developers first specify a general strategy for identifying matching units across hyperslices, and then defining exceptions to, or specializations of, that strategy for those cases where the strategy does not apply. Hyper/J supports three general strategies at present. These are called **mergeByName**, **nonCorrespondingMerge**, and **overrideByName**.

- **mergeByName** indicates that units in different hyperslices that have the same name are to *correspond*, and are to be connected by a *merge* relationship, which causes connected units to be integrated together into a new unit. This is the most commonly used strategy.

- **nonCorrespondingMerge** means that units in different hyperslices with the same name are *not* to correspond, and hence are not to be connected, by default, by any relationship. Non-corresponding merge is generally used in circumstances where units in different hyperslices accidentally have the same name, but are not actually related to one another.

- **overrideByName** indicates that units with the same name are to correspond, and are to be connected by an *override* relationship, which causes the last one to override the others in the composed software. The order is determined by the order of the hyperslices in the hypermodule specification: of the units related by any override relationship, the one that prevails is the one belonging to the hyperslice that occurs latest in the list. Overriding really only affects methods, indicating which is actually to be executed. If one class overrides another, for example, that does not mean that it totally replaces the other, just that it’s methods override corresponding methods in the other.

The relationships section of every hypermodule specification begins with one of these three strategies. They derive from the corresponding *composition rules* supported by subject-oriented programming, whose semantics are described more formally in [oss96].

The general strategy may or may not be sufficient to describe the relationships across hyperslices. If it is sufficient, no other relationships need be specified. If it is insufficient, other relationships can be described. The other relationships Hyper/J supports are defined in the remainder of this section. For each kind of relationship, we present the syntax by example, and then explain the relationship’s semantics.

4.2.3.4 Equate:

```
equate class SomeDimension.SomeConcern.SomeClass,
             SomeOtherDimension.SomeOtherConcern.SomeOtherClass;
```

```
equate operation Feature.Kernel.process,
                 Feature.Check.check_process,
                 Feature.Eval.eval_process,
                 Feature.Display.display_process
   into myProcess;
```

The **equate** relationship indicates that a set of units are to match each other—to correspond— even if their names are not the same. This relationship is used to set up correspondence *only*—the specific integration relationship connecting the corresponding units depends on the general composition strategy. For the first example above, if the general strategy selected was **mergeByName** or **nonCorrespondingMerge**, this **equate** relationship would cause the creation of a merge relationship between the two corresponding classes `SomeDimension.SomeConcern.SomeClass` and `SomeOtherDimension.Some-OtherConcern.SomeOtherClass`. If the general strategy was **overrideByName**, however, this **equate** relationship would cause the creation of an override relationship between these two classes; whichever belongs to the later-occurring hyperslice will override the other.

The **equate** relationship takes an optional **into** specification, which indicates the name that is to be given to the composed entity. The second example above demonstrates this feature. In this case, four operations—`Feature.Kernel.process`, `Feature.Check.check_process`, `Feature.Eval.eval_process`, and `Feature.Eval.eval_process` are to be equated. If the **mergeByName** strategy is used, this equate relationship will produce a composed operation named `myProcess`, which will be composed of these four operations. If no name is specified explicitly, Hyper/J synthesizes a name from the names of the related entities. If a user cares about the name of the composed entity, he/she should specify it explicitly.

Equate relationships can be applied to any kind of unit, but all equated units must be of the same kind. The general syntax for **equate** relationships is:

```
equateRelationship ::=
   equate unitKind unitName [, unitName]* ;
unitKind ::= class | interface | operation | action | field
```

The name given to the composed unit is synthesized by Hyper/J from the names of the input units. It can be changed, if desired, by means of **rename** (Section 4.2.3.6).

4.2.3.5 Order:

```
order action SomeDimension.SomeConcern.SomeClass.foo
before action SomeOtherDimension.SomeOtherConcern.SomeOtherClass.foo;
```

When methods are merged, Hyper/J can, by default, choose to run the code for the original methods in any order. Sometimes, however, the order is significant. For example, if one hyperslice provides some core functionality, while another defines some enhancements to the core functionality, the core hyperslice's methods should typically be run before the extension hyperslice's methods. The **order** relationship indicates that the order of related units is significant, and it describes any order constraints. Note that **order** relationships define *partial* orders—they indicate that one method must precede or follow another, but they need not fully specify the exact order in which to run the methods. Hyper/J will choose an order that satisfies all order constraints, if one exists (or report an

error if one does not exist). If an exact order is required, the developer need only specify enough **order** relationships to constrain the possible orderings to the one that is desired.

Although **order** relationships are intended to affect the ordering of composed methods, they can be applied to hyperslices, classes, interfaces, operations, and actions (but not to fields). When a hyperslice or class is used in an **order** relationship, it is simply a convenient shorthand for all of the methods defined within it. An operation is shorthand for all methods, in whatever hyperslices or classes, that implement that operation, and an interface is shorthand for all the operations within it.

The general syntax for **order** relationships is:

```
orderRelationship ::=
   order unitKind unitName [, unitName]* (before | after)
         unitKind unitName [, unitName]*;
unitKind ::= hyperslice | class | interface | operation | action
```

4.2.3.6 Rename:

```
rename class HypermoduleName.SomeClass to SomeNewName;
```

The **rename** relationship is not really a relationship, but rather, a directive to Hyper/J. It indicates that a specific unit in the composed hyperslice (which is referred to by the name of the hypermodule) is to be given a new name. In the above example, the composed software has a class named `SomeClass`, which the developer has asked to be renamed to `SomeNewName`.

Rename directives can be applied to any type of unit, but only to those that occur in the *composed* hyperslice. It is not legal to rename units in the input hyperslices with this directive (and it is not necessary, either).

The general syntax for **rename** directives is:

```
renameRelationship ::=
   rename unitKind unitName to newUnitName;
unitKind ::= class | interface | operation | action | field
```

4.2.3.7 Merge:

```
merge class SomeDimension.SomeConcern.SomeClass,
            SomeOtherDimension.SomeOtherConcern.SomeOtherClass;
```

The use of **merge** causes Hyper/J to create a merge relationship between the set of units that are specified, whether or not the units matched each other based on the general composition strategy. It differs from the **equate** relationship in that **equate** does not cause the equated units to be merged; it only indicates that the equated units correspond, with the relationship that is ultimately created among the equated units depending on the general composition strategy. The **merge** relationship causes the named units to be equated *and* merged, independent of the general composition strategy. The name given to the composed unit is the same as for **equate**.

Merge relationships can be applied to any kind of unit. The general syntax for **merge** relationships is:

```
mergeRelationship ::=
   merge unitKind unitName [, unitName]* ;
unitKind ::= class | interface | operation | action | field
```

4.2.3.8 NoMerge:

```
noMerge class SomeDimension.SomeConcern.SomeClass,
              SomeOtherDimension.SomeOtherConcern.SomeClass;
```

The **noMerge** relationship has the opposite effect to the **merge** (or **override**) relationship; it causes two or more units that match each other *not* to be merged (overridden), even if the general composition strategy is to merge (override) them. **noMerge** is typically used in cases where **mergeByName** or **overrideByName** is used as the general composition strategy, but where some units that match by name are not intended to be merged or overridden. In the above example, the classes `SomeDimension.SomeConcern.SomeClass` and `SomeOtherDimension.SomeOtherConcern.SomeClass` match by name, but the developer does not want these classes to correspond.

noMerge relationships can be applied to any kind of unit. The general syntax for **noMerge** relationships is:

```
noMergeRelationship ::=
   noMerge unitKind unitName [, unitName]* ;
unitKind ::= class | interface | operation | action | field
```

4.2.3.9 Override:

```
override action SomeDimension.SomeConcern.SomeClass.foo with
         action SomeOtherDimension.SomeOtherConcern.SomeClass.foo;
```

The **override** relationship indicates that one unit overrides one or more other units with which it corresponds, in the sense described for **overrideByName** earlier. In the above example, the method `SomeOtherDimension.SomeOtherConcern.SomeClass.foo` overrides `SomeDimension.SomeConcern.SomeClass.foo`, which means that anywhere the software refers to either of these methods, only `SomeOtherDimension.SomeOtherConcern.SomeClass.foo` will be invoked.

Although **override** relationships are intended to affect the definition of composed *methods*, they can be applied to any kind of unit. When an **override** relationship is specified for a unit other than an action, it is simply a convenient shorthand for indicating that when methods in one unit match those in another unit, the methods defined in the *last* unit specified are to override all the corresponding methods defined in the other units, as described earlier for **order**.

The general syntax for **override** relationships is:

```
overrideRelationship ::=
   override unitKind unitName [, unitName]* with
            unitKind unitName;
unitKind ::= class | interface | operation | action
```

4.2.3.10 Match:

```
match class SomeDimension.SomeConcern.SomeClass with "*";
```

The **match** relationship is used to indicate that a given unit should match a set of units,specified using pattern matching on unit names. For example, the **match** specification above indicates that the class `SomeDimension.SomeConcern.SomeClass` should match all other classes ("*").

Like **equate, match** relationships do *not* themselves cause the matched units to be integrated in any way, they just imply correspondence. Instead, the units they cause to be matched are still subject to the general composition strategy. For example, if **mergeByName** is used, then all matched units will be merged; if **overrideByName** is used, then one of the matched units will override the others.

Matching only occurs for same-typed units. Thus, for example, classes will only match classes, and instance variables will only match instance variables.

The syntax for match patterns is illustrated by example below.

`"foo"`	matches only those units named foo
`"foo*"`	matches those units whose names start with foo
`"foo*bar"`	matches units whose names start with foo and end with bar
`"~foo*"`	matches units whose names do *not* start with foo
`"~foo"`	matches any units except those named foo
`"{foo,bar}*"`	matches any units that start with either foo or bar
`"*{foo,bar}"`	matches units whose names end with either foo or bar
`"{f,~foo}*{~r}"`	matches units whose names start with f but do not start with foo, and that do not end with the letter "r"

4.2.3.11 Bracket:

```
bracket "*"."foo*"
   from action Application.Concern.Class.bar
   before Feature.Logging.LoggedClass.invokeBefore($ClassName),
   after Feature.Logging.LoggedClass.invokeAfter($OperationName);
```

The **bracket** relationship indicates that a set of methods should be *bracketed*—i.e., their invocation should be preceded and/or followed—by other specified methods. For example, in the **bracket** relationship above, all methods whose names begin with "foo" in any class in the input hyperslices will be bracketed by the methods `Feature.Logging.LoggedClass.invokeBefore` and `Feature.Logging.LoggedClass.invokeAfter`. Thus, when the composed software invokes a method called `foo()`, the call will result first in executing `invokeBefore()`, then `foo()`, then `invokeAfter()`.

A **bracket** relationship can also optionally include a *callsite specification* (the **from** clause above). A callsite specification is used to restrict the calling context from which the bracket methods will be invoked. For example, the **from** clause of the **bracket** relationship defined above indicates that the **before** and **after** methods should only be invoked when foo() methods are called from within the method `Application.Concern.Class.bar()`. If foo() methods are called from anywhere else, the **before** and **after** methods will not be invoked. Thus, the body of the callsite

specification identifies the program units where the **before** and **after** methods should be included in a call to the matched method. The callsite specification can reference any kind of unit. Callsites based on **action**, **operation**, and **class** units apply to the indicated Java structures. Callsite specifications that use a **hyperslice** unit can restrict bracketing to precisely defined software units. The `None` hyperslice can be used to include all unspecified software units in a hyperspace dimension.

Bracket relationships require several pieces of information to be specified, as shown above:

- First, developers must indicate the set of operations in the hypermodule that they want to be bracketed. These are specified by pattern matching (see the description of the **match** relationship in this chapter for details of the match pattern syntax). Developers may specify either just a pattern for an operation name (e.g., "foo*"), in which case all operations with that name—irrespective of the class in which they appear—will be bracketed, or they may specify a pattern for a class name and one for the operation name, to restrict the set of methods that will be matched (e.g., "C*"."foo" would bracket only those foo() methods appearing in classes whose name starts with C).

- Next, they must indicate the "before" and/or "after" methods that are to bracket the indicated methods. Note that developers may specify only a "before" method or only an "after" method, as appropriate for the particular circumstances.

- Finally, developers must specify a *class match pattern*. If it is "*", a common case, all methods matching the operation match pattern ("foo*" in the example) will be bracketed. Otherwise, only those methods in classes that match the class match pattern will be bracketed.

In some cases, the bracket methods, like `invokeBefore` and `invokeAfter` above, may require parameters to be specified. If the bracket methods' parameter types are the same as those of the methods they bracket, then the same parameter values can be passed to the bracketed method and the bracket methods. In some cases, however, the bracket methods may require information about the methods that they are bracketing. At present, Hyper/J supports two pieces of information about the bracketed method: its operation name, and its class name. If this information is required by a bracket method, it can be specified using the (case-insensitive) keywords **$OperationName** and **$ClassName**, respectively, as shown in the example above.

The **bracket** relationship only applies to operations. The general syntax for **bracket** relationships is:

```
bracketRelationship ::=
   bracket [classMatchPattern .] operationMatchPattern
      [from unitKind unitName [, unitName]* ]
      [with]
      [before fullyQualifiedMethodName,]
      [after fullyQualifiedMethodName,]
unitKind ::= hyperslice | class | operation | action
```

Notes:

- Current limitations of the **bracket** relationship are that (a) you can specify only one **from** clause per bracket, though it can name several units; (b) you can only indicate one kind of unit per **from** clause; and (c) you may only define one **bracket** relationship involving any given **before** and **after** methods.

- It is an error for a **bracket** relationship to fail to match any operations.

4.2.3.12 Summary function:

```
set summary function for action DemoSEE.NumberLiteral.check
                             to DemoSEE.Expression.summarizeCheck;
```

When methods that return values are merged, the composed method must return just one value. Yet, each of the methods of which it is composed potentially will return a different value. What value does the composed method return?

By default, Hyper/J will return the value returned by the *last* of the methods of which it is composed. This may be appropriate in many cases. In other cases, however, the composed method should *synthesize* a return value based on some or all of the values returned by the methods of which it is composed. In this case, Hyper/J permits the developer to specify a *summary function*, which takes as input an array of values that were returned by sub-methods, and uses them to compute a single return value.

To illustrate summary functions, consider the example above. The `check()` method in the composed `NumberLiteral` class (`DemoSEE.NumberLiteral.check`) is composed of the `check()` methods defined in the Check and StyleChecker features (`Feature.Check.NumberLiteral.check` and `Feature.StyleChecker.NumberLiteral.check`, respectively). Each of these sub-methods returns a boolean value, which indicates whether or not a given sub-expression is syntactically or stylistically correct (respectively). The composed method should therefore return **true** if and only if *both* of the check sub-methods return true. If either returns **false**, then the composed method should return **false**.

To achieve this effect, the developer simply writes the following summary function:

```
static void summarizeCheck ( boolean[] returnResults ) {
   for ( int i = 0;
         i < returnResults.length;
         i++ )
      if ( !returnResults[i] )
         return false;
   // If we reach this point, all returnResults were true.
   return true;
}
```

(S)he then uses the summary function relationship to attach this method as a summary function to the appropriate composed `check()` method(s), as shown above.

Summary functions can be specified for specific methods (actions) or for operations. When specified for an operation, the summary function will be attached to *all* composed methods with the given operation name.

Hyper/J permits developers to name any *external* method—i.e., any Java static method, whether or not it is included in the hyperspace—as a summary method. To name an external method as a summary method, use the **external** keyword before the name of the summary function. For example,

```
set summary function for action DemoSEE.NumberLiteral.check
                    to external mySummaryFunctions.summarize;
```

In this case, the summary function mySummaryFunctions.summarize need not be in the hyperspace. If the **external** keyword is not present, Hyper/J expects to find the summary function in the hyperspace.

Hyper/J comes with a library of useful summary functions. These are defined in the package `com.ibm.hyeprj.SummaryFunctions` (source code is included in the `src` directory of the Hyper/J release). If you do not find the summary function you need, you can write your own.

The syntax of the summary function relationship is as follows:

```
summaryFunctionRelationship ::=
   set summary function for unitType unitName to summaryFunction;
summaryFunction ::= external unitName
                  | unitType unitName
unitType ::= action | operation
```

4.2.3.13 Notes:

- Summary functions are required to be *static* methods. They can have any visibility (public, private, protected, or package).
- Due to a current limitation, summary functions can only be attached to *actions*, and not to *operations*. The same effect can be achieved using just actions, but it may take more summary function relationship specifications to do so.
- Summary functions, and the actions or operations to which to attach them, must be defined in the *composed* hyperslice, and *not* in the input hyperslices. (Recall that the composed hyperslice always has the same name as the hypermodule.)

4.2.4. Using the Simplifications

In this section, we have noted several simplifications that Hyper/J offers developers: using a single control file (Section 4.1.1.4) to specify options to Hyper/J; omitting the hyperspace specification and allowing Hyper/J to derive it automatically from the concern mappings; and using the abbreviated integration relationship specification in place of a full hypermodules specification. To see how a developer might leverage these options to simplify the use of Hyper/J, consider a common scenario in which a developer has produced an extension of an existing system. In this case, the developer need only define two concerns—the existing system and the extension—and one integration relationship (mergeByName), since (s)he carefully used the same class, method, and instance variable names in the extension as were present in the existing system. This developer could write the following control file (which (s)he might name `simple.opt`), which employs all of the simplifications:

```
-concerns
   package ExistingSystem : Feature.ExistingSoftware
   package Extension : Feature.Extension

-relationships
   mergeByName
```

The developer could then run Hyper/J with the command

```
java com.ibm.hyperj.hyperj simple.opt
```

This is considerably easier and shorter than defining complete hyperspace and hypermodule specifications. For many development scenarios, these simplifications can reduce the time developers spend writing Hyper/J control information.

4.2.5. Unparsed Hyperslice Files

At times, it may be somewhat difficult to visualize what the composed software will look like, based solely on a hypermodule specification. To aid developers in understanding both what the composed hyperslice actually contains, and to help developers to identify errors in their composition relationships, Hyper/J can optionally produce *unparsed hyperslice files*. Unparsed hyperslice files contain a user-readable representation of hyperslices. They are created if a developer runs Hyper/J with the command-line options `-verbose` or `-debug`. Unparsed hyperslice files are generated into files whose names are the name of the corresponding hyperslice, with the suffix ".unp". One unparsed hyperslice file is generated for each of the input hyperslices in the hypermodule, and one is generated for the composed hyperslice (whose name is the same as that of the hypermodule).

The unparsed hyperslice file is structured as follows:

hyperslice *hypersliceName*

operations
list of all composable operations in the hyperslice

interfaces
alphabetical list of all composable and uncomposable interfaces in the hyperslice

classes
alphabetical list of all composable and uncomposable classes in the hyperslice

named types
Not used in Hyper/J

mapping
alphabetical list containing all of the composable operations in the hyperslice (taken from the composable operations section); for each operation, each method that implements the operation in all of the composable classes is listed; in other word, the method mapping

We describe the contents of each of these sections below. The examples are drawn from the expression SEE example described in detail in Section 5; the reader might wish to defer detailed reading of this section until after reading about, and running, the example.

4.2.5.1 Operations:

The operations section contains a list of all the *operations* defined in a given hyperslice. As described earlier, Hyper/J distinguishes *operations* from *methods* or *actions*, in that methods are specific *implementations* of operations that are defined within specific classes. Operations are simply defined by their name and signature. If two different classes implement methods named foo, which takes no parameters and returns void, there will be one operation with this name and signature in the operations section. The **mapping** section describes how each class in the hyperslice implements a given operation (if it does).

Operations in the **operations** section have the form:

```
setValue
   signature: (Expression newValue) returning void
```

This example describes an operation named `setValue`, which takes one parameter (`newValue` of class `Expression`) and returns `void`..

Notes:

- You may notice operations named `<init>` or `<clinit>` in the **operations** section. These are the names used in Java class files to represent class constructor methods and static initializers, respectively.

4.2.5.2 Interfaces:

The interfaces section provides information about all of the interfaces that are part of the hyperslice, composable or not. For each *composable* interface that is declared part of the hyperslice, this section indicates the interface's name and all of the operations defined within it. For each *uncomposable* interface, it indicates just the name of the interface; details are not gathered or reported by Hyper/J, since they are not needed for interfaces not involved in composition. For example:

```
interface Observer[Package: "demo.Observer"]
      inheritance parents:
         GenericObserver[Package: "demo.Observer"]
      operations:
         _acceptNotification(java.lang.Object[]) returning void

UNCOMPOSABLE interface Enumeration[Package: "java.util"]
```

In this case, an interface named `Observer` (defined in package `demo.Observer`) is part of the hyperslice. This interface is a sub-interface of another interface, `demo.Observer.GenericObserver`. It contains one operation, called `_acceptNotification`. Notice that the standard Java interface `Enumeration` is also defined to be part of this hyperslice, where it is being treated as uncomposable, so the unparsed hyperslice file does not list any operations for it.

4.2.5.3 Classes:

The classes section provides information about all of the classes that are part of the hyperslice, composable or not. For each *composable* class that is declared part of the hyperslice, this section indicates the class's name and all of the instance variables defined within it. It does *not*, however, include any operations or static (class) variables that are defined within the class. The set of operations that each class implements is shown in the **mappings** section instead. For historical reasons, the static variables for *all* classes are listed separately in the last part of the **classes** section, under the heading **class variables**. For each *uncomposable* class, the unparsed hyperslice file indicates just the name of the class. To illustrate:

```
class UnaryPlus[Package: "demo.ObjectDimension"]
      attributes: public
      default classification:
         UnaryOperator[Package: demo.ObjectDimension"]
      inheritance parents:
         UnaryOperator[Package: "demo.ObjectDimension"]
      instance variables:
         _operand
            type: Expression[Package: "demo.ObjectDimension"]
```

This indicates that a class named `UnaryPlus`, which is defined in a Java package called `demo.ObjectDimension`, is part of this hyperslice. The statement "**attributes: public**" indicates that class `UnaryPlus` is a public class. The **default classification** for a class in Java is always the superclass; in this case, that class is `UnaryOperator`, which is also defined in package `demo.ObjectDimension`. A class will always have exactly one superclass; this class will also be listed under the **inheritance parents** heading. Finally, this example indicates that class `UnaryPlus` has one instance variable, which is called _operand, and its type is class `Expression` (which is also defined in package `demo.ObjectDimension`). In this release, the attributes and inheritance parents are shown for input hyperslices only.

As noted above, the static variables (class variables) for *all* the classes in a hyperslice are listed separately in the last part of the **classes** section, under the heading **class variables**. An example of what might appear in the class variables subsection is:

```
class variables:
      _count[class: Driver[Package: "demo.ObjectDimension"]]
         type: int
         attributes: private
      _logStream[class: Logger[Package: "demo.Observer"]]
         type: PrintStream[Package: "java.io"]
         attributes: private
      _logger[class: Globals[Package: "demo.Observer"]]
         type: Observer[Package: "demo.Observer"]
         attributes: private
```

The above example indicates that three classes in this hyperslice define static variables: class `demo.ObjectDimension.Driver`, which defines the private int variable _count; class `demo.Observer.Logger`, which defines the private static variable named `_logStream` of type `java.io.PrintStream`; and class `demo.Observer.Globals`,

which defines the private static variable `_logger` of type `demo.ObserverWithInterfaces.Observer`.

4.2.5.4 Named Types:

This section is not currently used by Hyper/J.

4.2.5.5 Mapping:

As noted earlier, Hyper/J distinguishes between *operations*, which simply include names and signatures, and *methods* or *actions*, which are specific implementations of operations for specific classes. The **mapping** section of an unparsed hyperslice file describes this mapping from operations and classes to actual implementations (actions). If a given class does not implement some operation that appears in the **operations** section, the **mapping** section does not include an entry for that <*operation, class*> pair.

The **mapping** section is organized by operation, with the operations appearing in alphabetical order. The following example illustrates the contents of the **mapping** section for the operation `getOperand`, which takes no parameters and returns class `demo.ObjectDimension.Expression`. In this case, there are three classes that implement a `getValue` method with this signature: classes `demo.ObjectDimension.UnaryMinus`, `demo.ObjectDimension.UnaryPlus`, and `demo.ObjectDimension.UnaryOperator`.

```
getOperand[signature: "() returning Expression[
                                    Package:"demo.ObjectDimension"]"]
      class UnaryMinus[Package: "demo.ObjectDimension"]
            inherited Compound action
            Label getOperand:
               inherited Simple action getOperand

      class UnaryOperator[Package: "demo.ObjectDimension"]
            Compound action
            Label getOperand:
               Simple action getOperand


      class UnaryPlus[Package: "demo.ObjectDimension"]
            inherited Compound action
            Label getOperand:
               inherited Simple action getOperand
```

In the above example, the notation **inherited Compound action** means that classes `UnaryPlus` and `UnaryMinus` inherit their implementations of the `getOperand` operation from their superclass (which is `UnaryOperator`). No other classes in this hyperspace implement a `getOPerand` operation with this signature.

The above example came from an *input* hyperslice. Mappings in composed hyperslices may be slightly more complex; for example, consider just the `UnaryPlus` composed class:

```
   getValue[signature: "() returning Expression"]
       class UnaryPlus
```

```
            CallAction: Sequence Java[return final value]
            Label Feature___Logging._beforeInvoke:
               CallAction: Simple Java[apply operation
   Feature___Logging._beforeInvoke(String[Package: "java.lang"],
   String[Package: "java.lang"]) returning void]
               Label className:
                  Class name accessor
               Label methodName:
                  Operation name accessor
            Label Feature___Kernel.getValue:
               CallAction: Simple Java[apply operation
   Feature___Kernel.getValue() returning Expression[Package:
   "demo.ObjectDimension"]]
            Label Feature___Logging._afterInvoke:
               CallAction: Simple Java[apply operation
   Feature___Logging._afterInvoke(String[Package: "java.lang"],
   String[Package: "java.lang"]) returning void]
               Label className:
                  Class name accessor
               Label methodName:
                  Operation name accessor
```

In this case, the implementation of `getValue` in class `UnaryPlus` has the annotation **CallAction: Sequence Java[return final value]**. This means that the composed method comprises multiple input methods. In particular, the above mapping indicates that the getValue method in the composed UnaryPlus class is composed of the methods `Feature.Logging._beforeInvoke`, `Feature.Kernel.getValue`, and `Feature.Logging._afterInvoke`, in that order. (This is the expected result from using the **bracket** relationship described earlier in this section.)

Notice the description of _beforeInvoke above:

```
   Label Feature___Logging._beforeInvoke:
               CallAction: Simple Java[apply operation
   Feature___Logging._beforeInvoke[(String[Package: "java.lang"],
   String[Package: "java.lang"]) returning void]
               Label className:
                  Class name accessor
               Label methodName:
                  Operation name accessor
```

In particular, the last four lines, **Class name accessor** (`className`) and **Operation name accessor** (`methodName`), indicate that when invoking `_beforeInvoke` as part of this composed method, the names of the composed class and method will be passed as parameters to `_beforeInvoke`.

Notes:

- For historical reasons, all static methods are shown in the **mapping** section under **class static**, rather than under the actual classes in which they were defined.

4.3. Designing Using Hyper/J

One of the benefits of Hyper/J is that it permits developers to design their software so that the structure of the software reflects all of their concerns of interest—i.e., to separate any concerns of importance from the start. This section provides some information for developers who are using Hyper/J to keep their concerns separate from the start.

4.3.1. Developing with "Hyperslice Packages"

The code for each concern should be written completely separately from the code that implements other concerns. To do this in Java™, developers should implement each concern in its own, separate Java package (or packages). We call such a package a *hyperslice package*, because it is deliberately written to encapsulate a concern. It can then be integrated with other concerns as needed. This adds tremendous flexibility to the code architectures that developers can select, and to the range of software development processes they can use, since the classes, interfaces, and members contained within each hyperslice package can *overlap* those of other hyperslice packages. Further, the class structures in different hyperslice packages can differ from one another somewhat, even if these hyperslice packages are intended to be integrated into a single system. An example of development with hyperslice packages is given in Section 5.3.1.

While Hyper/J can be used at any stage of the software lifecycle—from early design to evolution—we strongly encourage developers to use it as early in the lifecycle as possible. Separating concerns up-front can greatly simplify the initial design of a software system, which results in greater evolutionary flexibility as well.

Hyperslice packages must be written in standard Java, and they must be compiled successfully (by any Java compiler) before they can be input to Hyper/J.

4.3.2. Using Declarative Completeness in Hyperslice Packages

As noted above, every hyperslice package must be compilable. Yet any given concern may expect other concerns to define some of the capabilities on which it depends. Section 3.5.3 described Hyper/J's *declarative completeness* requirement: i.e., within a hyperslice, every unit to which a given unit refers to must, at minimum, be *declared* within that hyperslice. Java imposes the same declarative completeness requirement; thus, hyperslice packages must be declaratively complete.

When writing hyperslice packages, therefore, it is often necessary to declare methods whose implementations will occur in some other hyperslice package, to allow standard Java compilation. We recommend the following strategies for ensuring declarative completeness:

- If possible, declare the methods as "abstract" in Java. This clearly marks this method as being one that is *required* within this hyperslice, but not *defined*, and which the hyperslice package developer expected to be provided by another hyperslice.

  If you employ this approach, you will also have to declare the class in which the abstract method appears as "abstract." This strategy works well and is the simplest solution, but it can only be used for classes that are not instantiated within the hyperslice package, since abstract classes cannot be instantiated. (Note that when

an abstract class, C1, in one hyperslice is composed with a class, C2, in another hyperslice, where C2 provides implementations for C1's abstract methods, the composed class will actually be concrete.)

- If the "abstract" strategy is either not feasible (because the resulting abstract class must be instantiated within the hyperslice) or not desirable, you may provide a "dummy" implementation for needed methods, which marks them as required by this hyperslice package but not defined within it. You should *always* use the following "dummy" implementation for any method defined for declarative completeness purposes:

```
throw new com.ibm.hyperj.UnimplementedError();
```

  Hyper/J recognizes this implementation, and it treats any method implemented this way as an abstract method. When such an "unimplemented" method is composed with a hyperslice that provides a real implementation, the "dummy" implementation is discarded, and only the real implementation is used in the composed output. If methods that are composed together contain only this "dummy" implementation, Hyper/J will generate the composed method to throw the unimplemented error if it is invoked.

4.4. Causes of Common Hyper/J Error Messages

Hyper/J attempts to diagnose and report possible causes of errors whenever possible. If you encounter errors and cannot determine the cause, please send mail to hyperj-support@watson.ibm.com, and a member of the Hyper/J team will assist you. Please include in your message the full error message; a team member may ask you for additional information, based on the particular message.

A few errors occur fairly commonly. We list them, and their likely causes, below.

com.ibm.sop.util.SOPInternalError: SOP internal error: Could not find java.lang.Object in hyperslice <*some hyperslice name*>; see dump in <*hyperslice name*>_ERROR

OR

com.ibm.sop.hyperspace.HypersliceExtractionError: Error extracting hyperslice: <*some hyperslice name*> does not contain any units. The most likely cause is an erroneous concern mapping, a hyperspace specification that mentions non-existent classes, or a misset CLASSPATH

Both of these errors indicate that the named hyperslice was defined in such a way as to have no classes. It is not legal to have empty hyperslices. In general, these errors occur because the developer did not specify a concern mapping correctly, or because some class files that the developer intended to include in the hyperspace were not found. To determine if this is the case, use the `-verbose` option to Hyper/J and check the hyperspace dump file (BeforeCompositionHyperspace.dump) and/or the hyperslice unparse file(s) to find out which classes were actually loaded. Compare this list with the set of classes you intended to have loaded. If all the classes were loaded, the problem is that your concern mapping did not map anything to the

concern corresponding to the named hyperslice. Just add the appropriate mappings. If classes are missing, do the following:

- Check your hyperspace specification file to be sure you included the missing classes. If you specified the class files by file name, check the locations. If you specified the classes by fully qualified Java class name, check the classes to be sure that they are actually defined in the package you indicated.

- Check your class path to be sure that it includes the directories needed to find the classes.

- Check the directories from which the missing class files should have been loaded to ensure that the class files are actually there.

Warning: class *<first class name>* **has no constructor to compose with operation** *<operation name>* **of class** *<second class name>* **et al**

This message is generally harmless. It indicates that a class named *<first class name>* in one hyperslice was matched with *<second class name>* in a different hyperslice, and *<second class name>* has a constructor named *<operation name>*, but *<first class name>* does not have any constructor with compatible parameters. The default constructor, with no parameters, can be composed with any constructor, so this message should only occur if a concern mapping has divided a single Java class into multiple concerns, and the constructors (especially the default constructor) were associated with other concerns, or in the case of classes with no default constructors. Unless you depended on having *<first class name>*'s constructor composed with *<second class name>*'s constructor, you can safely ignore this message.

java.lang.NoSuchMethodError

This error typically occurs for users of Sun JDK 1.2 or higher. If it does occur, you probably need to add the following JDK files to your CLASSPATH (if they are not already there):

```
%JAVA_DIR%\lib\tools.jar
%JAVA_DIR%\jre\lib\rt.jar
%JAVA_DIR%\jre\lib\jaws.jar
%JAVA_DIR%\jre\lib\i18n.jar
```

4.5. Current Hyper/J Limitations and Known Problems

Hyper/J has several current limitations on the functionality described in this chapter. The set of currently known limitations and problems is listed below; it is our intention to address these limitations as soon as possible. If you encounter other limitations that are not listed, please report them to hyperj-support@watson.ibm.com. When sending the report, please include a specific example, if possible. Similarly, if you find that the existing functionality is not sufficient for your needs, please let us know.

- Error reporting may be somewhat inconsistent. If you encounter difficulties in understanding error messages, please contact hyperj-support@watson.ibm.com for assistance.

- The Java predefined classes, like `Object` and `String`, cannot currently be composed, though they can be included in hyperspaces as uncomposable classes.

- The **noncorrespondingMerge** general composition strategy does not currently work correctly and has been disabled.

- The **merge** and **override** composition relationships currently do not work; mergeByName and overrideByName do work as general composition strategies applying to entire hyperslices.

- The **noMerge** composition relationship works only on operations at present.

- The **order** relationship only works correctly at present when it is specified for *actions*. Developers can achieve the effect of hyperslice-level ordering by listing the hyperslices in the **hyperslices** section of the hypermodule specification file in the order desired.

- Summary functions must be *static* methods, and they must be defined in the *composed* hyperslice.

- Match pattern relationships can only be specified on operations and classes. They do not work on the special class **static**, and "*" does not include **static**.

- The class match pattern in bracket relationships also does not work on the special class **static**, and "*" does not include **static**. Static methods cannot, therefore, currently be bracketed.

- In hyperspace specifications, the wildcard syntax does not currently cause the loading of classes in all sub-packages. Developers should specify explicitly the inclusion of any sub-packages.

- Only one hypermodule can be defined in each hypermodule specification file.

- All classes, interfaces and members in the composed hyperslice are public, irrespective of the visibility modifiers in the input class files.

- All classes in the composed hyperslice are in a single, top-level package, irrespective of package structure in the inputs.

Chapter

5

Software Development and Evolution using Hyper/J: An Example

To illustrate the use of Hyper/J and convey a sense of some of the different ways in which developers can leverage its capabilities throughout the software development lifecycle, we present here part of the development process of the expression SEE introduced in Chapter 3. Only small illustrations of code are shown here; the full, runnable code for the SEE example is included in full in the Hyper/J release, in directory "demo".

Important note for running the example:

- The class path you will use when running Hyper/J will conflict with the class path you need to use to run your composed programs. This is because the composed programs generally contain classes with the same names as the classes you input to Hyper/J. To avoid problems stemming from using the wrong class path, we recommend setting up two shell windows: one in which to run Hyper/J, and one in which to run composed programs. On a Windows system, set up the class paths for running the demo examples as follows:

  Be sure that %HYPERJ_DIR% refers to the Hyper/J installation directory in both the following Shell Windows, then:

  In Shell Window 1 (to run the original example and Hyper/J):
  SET CLASSPATH=%HYPERJ_DIR%;%HYPERJ_DIR%\bin\hyperj.jar;%CLASSPATH%

  In Shell Window 2 (to run the composed program produced by Hyper/J):
  SET CLASSPATH=.\DemoSEE;%HYPERJ_DIR%\bin\hyperj.jar;%CLASSPATH%

  It is, of course, possible to use explicit `-classpath` parameters when running Java, but doing so is less convenient and more error-prone.

5.1. 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™, to produce the design shown in *Figure 2*. The code is in a single package, demo.ObjectDimension.

Feature concerns are not identified or encapsulated within this code. The Check, Display and Evaluation features are all present, in addition to the Kernel. When the example SEE is executed (do this in **Shell Window 1**; see the note in the introduction to this section) with the command

```
java demo.ObjectDimension.Driver
```

the output includes output from all the features.

The mechanism by which all the features are executed in the example SEE is the process() method. This method, defined on Expressions, is called by the Driver on each expression it works with. It is implemented in demo\ObjectDimension\Expression.java as follows:

```
1. public void process()
2.  {
3.     System.out.println ( "Beginning expression processing..." );
4.
5.     // Notice that the order and content are hard-coded here:
6.     check_process();
7.     eval_process();
8.     display_process();
9.  }
```

As noted in the comment, the calls to feature-specific process methods, and their order of execution, are hard-coded, in lines 6 to 8.

5.2. Mix-and-Match of Features (and Developing Product Lines)

The first change in the requirements entailed permitting the creation of different versions of the expression SEE, each with different subsets of features. Hyper/J™ can help here in two ways.

- It provides on-demand remodularization—the ability to identify and encapsulate new dimensions of concern at any time, without invasive changes. Thus, developers can introduce the needed feature concerns using Hyper/J™, and then manipulate those concerns as first-class entities.

- Hyper/J's composition capability permits the *selective* integration of concerns, and hence creation of variants of the expression SEE that integrate different subsets of the available features, as needed, non-invasively.

To use Hyper/J™ to accomplish this task, a developer performs the steps described in the following sections.

5.2.1. Create a Hyperspace Specification File

The file demo\ObjectDimension.hs lists all the classes that make up the expression SEE, thereby specifying all the units to be brought into the hyperspace:

```
hyperspace DemoHyperspace
   composable class demo.ObjectDimension.*;
```

This simple file specifies that all classes within the package demo.ObjectDimension should be included.

When Hyper/J™ runs, it will automatically create one dimension—the *ClassFile* dimension—and one concern in that dimension for each class file it loads. The contents of those concerns are the units (interfaces, classes, methods, and member variables) in the corresponding class files.

5.2.2. Create Concern Mappings

To achieve the mix-and-match of features that is desired, the developer must first encapsulate the features as first-class concerns. S/he does this by creating a new dimension—the Feature dimension—and describing how existing units in the hyperspace address concerns in that dimension. To do so, s/he specifies *concern mappings* in the concern mapping file demo\ObjectDimension\concerns.cm:

```
package demo.ObjectDimension : Feature.Kernel
operation check              : Feature.Check
operation display            : Feature.Display
operation eval               : Feature.Eval
operation check_process      : Feature.Check
operation display_process    : Feature.Display
operation eval_process       : Feature.Eval
operation process            : Feature.None
```

The first mapping indicates that, by default, all of the units contained within the Java™ package demo.ObjectDimension address the Kernel concern in the Feature dimension. Since the Feature dimension does not yet exist when Hyper/J™ processes this first concern mapping, Hyper/J™ will create it (and the Kernel concern). The next three mappings indicate that any methods named “display,” “check,” or “eval” address the Display, Check, or Eval features, respectively. The following three mappings are similar. These later concern mappings override the first one, whenever they apply. This illustrates an approach employed throughout Hyper/J™: specification of a general rule followed by exceptions, to clarify and shorten specifications.

The final concern mapping relates to the Expression.process() method, mentioned earlier:

```
1. public void process()
2.  {
3.     System.out.println ( "Beginning expression processing..." );
4.
5.     // Notice that the order and content are hard-coded here:
6.     check_process();
7.     eval_process();
8.     display_process();
9.  }
```

We would like to say that line 6 belongs to the Check feature, line 7 to the Eval feature and line 8 to the Display feature. Hyper/J currently treats methods as primitive units, however, which means that it does not support such mapping of individual statements to concerns. Since we can’t pull this method apart, we need to exclude it entirely to achieve mix-and-match of features. We’ll see later how to use Hyper/J composition instead to invoke the features we desire. To exclude this method, we map it to Feature.None, thereby declaring that it belongs to no feature.

Once Hyper/J™ has processed these concern mappings, the concern matrix will contain two dimensions: ClassFile and Feature. Each unit addresses exactly one concern in each dimension. Thus, for example, the method Expression.display() addresses the concern demo.ObjectDimension.Expression in the ClassFile dimension, and the Display concern in the Feature dimension.

5.2.3. Create a Hypermodule Specification File

Once the feature concerns have been identified, the developer can create versions of the SEE that contain different sets of features by defining *hypermodules*. For example, the following hypermodule specification file, demo\CheckDisplay.hm, defines a version of the SEE that contains the Kernel, Check and Display capabilities only:

```
hypermodule DemoSEE
   hyperslices:
      Feature.Kernel,
      Feature.Check,
      Feature.Display;
   relationships:
      mergeByName;
      equate operation Feature.Kernel.process,
                       Feature.Check.check_process,
                       Feature.Display.display_process;
end hypermodule;
```

In this hypermodule, the Kernel, Check and Display concerns are related by a "mergeByName" integration relationship. The "ByName" indicates that units in the different concerns are considered to correspond if they have the same names (and signatures, where appropriate). The "merge" indicates that corresponding entities are to be combined so as to include all their details; for example, all members in corresponding classes are brought together in the composed class.

The second integration relationship, "equate," accomplishes the special handling of the process() method. As discussed earlier, we excluded process() from the hypermodule by relegating it to the Feature.None concern. However, the Driver calls it, within the Feature.Kernel concern. During declaration completion, to make Feature.Kernel a valid hyperslice, Hyper/J™ inserts an abstract declaration of process(). In this hypermodule, we want to specify that that abstract declaration be bound to both check_process() from the Check feature and display_process() from the Display feature. The "equate" relationship does just that. It ensures that, when the Driver calls process() at run time in the composed hyperslice, both check_process() and display_process() will in fact be called.

The hyperslice that results from composing these concerns will contain all the AST classes, but with just Kernel, Display and Check functionality in each. In particular, no eval() methods will be present.

5.2.4. Run Hyper/J™

Once the three files described above have been written, it its time to run Hyper/J™. The current directory can be any directory in which files can be written. Use the following commands in **Shell Window 1** (see the note in the introduction to this section), noting that the "java" command should be all one line; layout here is for ease of reading:

```
java com.ibm.hyperj.hyperj
   -hyperspace   %HYPERJ_DIR%/demo/ObjectDimension.hs
   -concerns     %HYPERJ_DIR%/demo/ObjectDimension/concerns.cm
   -hypermodules %HYPERJ_DIR%/demo/CheckDisplay.hm
   -verbose
```

The class path must include the Hyper/J release directory, because that is where the example "demo" directory is located, and Hyper/J uses the class path to locate all the class files it reads.

This will produce the "composed hyperslice:" a collection of Java™ class files for the composed classes, produced by integrating the input hyperslices as specified by the integration relationships. These files will be in the directory DemoSEE, created (or reused) within the current directory and named as specified in the hypermodule specification file, CheckDisplay.hm. The directory will also contain a pseudo-source (.java) file corresponding to each class file, for use with debuggers; these files are not full Java™ for the composed classes, however, and cannot be compiled.

The "-verbose" option will cause some messages and a number of files to be produced that are useful aids to understanding Hyper/J™ and debugging the composition. Most important are the unparsed hyperslice files for the input hyperslices (the features) and the composed hyperslice, DemoSEE. The nature of these files was described in Chapter 4. We recommend that the reader examine them briefly after running this example to get a concrete feel for them.

5.2.5. Run the Composed Hyperslice

To run the variant of the SEE created above, execute the class files in the composed hyperslice in **Shell Window 2** (see the note in the introduction to this section):

```
java demo.ObjectDimension.Driver
```

The current directory, which contains the DemoSEE directory containing the composed hyperslice, must be on the class path. The original Java™ composable classes need not be on the class path at all, but any library or other non-composable classes used by the composable classes must be.

This execution produces output from just the Check and Display features. Comparison with the output of the original program shows absence of the results produced by the Eval feature.

The class files in DemoSEE contain standard debugging tables, referring to the generate pseudo-source (.java) files. The composed hyperslice can therefore also be debugged with debuggers that uses the standard tables. Details of how to do this depend on the debugger.

5.2.6. Summary

This part of the SEE evolution scenario has demonstrated the utility of Hyper/J's on-demand remodularization and integration capabilities on *existing* code. Notice that the feature concerns did not have to be identified or separated during initial development to permit them to be encapsulated. Also notice that each of the concerns is itself 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.

5.3. The Addition of Style Checking

The expression SEE clients eventually requested an enhancement that permits optional style checking of expression programs. Hyper/J™ allows the new feature to be developed separately from the existing features, and non-invasively, without modifying any of the existing code. The steps are described in the following sections.

5.3.1. Write and Compile a "Hyperslice Package"

To keep the new feature completely separate from the existing code, it should be written as a new, separate Java™ package (or packages). We call such a package a *hyperslice package*, because it is deliberately written to encapsulate a concern. It will then be integrated with other concerns as needed. This adds tremendous flexibility to the code architectures that developers can select, and to the range of software development processes they can use.

Figure 3 depicts the design of the new hyperslice package that realizes the style checking feature. Notice that the package contains *solely* the code needed to implement the style checking feature (plus abstract declarations, not shown, for anything "foreign" that is used, such as accessor methods from Kernel). Its class structure is similar to that of the original system (*Figure 2*), but not identical, because style checking only affects some of the Expression classes. This is an important feature of multi-dimensional separation of concerns using Hyper/J™: 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 the appropriate relationships between the concerns.

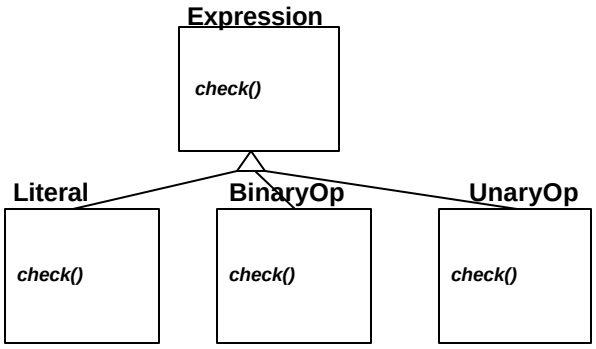

Figure 3. The Style Checking Hyperslice Package

The Java™ code corresponding to *Figure 3* is written in directory demo\StyleChecker. Note that it is absolutely standard Java™. Before it can be integrated by Hyper/J™, it must be compiled, by any Java™ compiler.

5.3.2. Create a Hyperspace Specification File

A new hyperspace specification file, demo\StyleChecker.hs, is needed, to include both the original class files and those making up the StyleChecker hyperslice package:

```
hyperspace DemoHyperspace
 composable class demo.ObjectDimension.*;
 composable class demo.StyleChecker.*;
```

5.3.3. Create Additional Concern Mappings

Inclusion of the StyleChecker hyperslice module above automatically specifies new concerns in the ClassFile dimension, but not in the feature dimension. A simple concern mapping is needed to create a StyleChecker feature concern, and to map everything in the StyleChecker hyperslice to it:

```
package demo.StyleChecker : Feature.StyleChecker
```

This concern mapping is in file demo\StyleChecker\concerns.cm.

5.3.4. Create a Hypermodule Specification File

With the StyleChecking feature now identified as a concern, the developer can create variants of the expression SEE that contain style checking or not, as desired, in much the same way as s/he can mix-and-match the other features, described earlier. For example, the following hypermodule specification file, demo\CheckDisplayStyle.hm, defines a version of the SEE that contains the Kernel, Check, Display and StyleChecking capabilities only:

```
hypermodule DemoSEE
   hyperslices:
      Feature.Kernel,
      Feature.Check,
      Feature.Display,
      Feature.StyleChecker;
   relationships:
      mergeByName;
      equate operation Feature.Kernel.process,
                       Feature.Check.check_process,
                       Feature.Display.display_process;
      set summary function for action DemoSEE.BinaryOperator.check
          to action DemoSEE.Expression.summarizeCheck;
end hypermodule;
```

This hypermodule specification is identical to the one we wrote before, except that we have added Feature.StyleChecker to the list of hyperslices, and there is an additional “set summary function” integration relationship at the end. To understand this new relationship, it is helpful to consider what mergeByName will do now that the StyleChecker hyperslice has been included. Examination of *Figure 3*, or of the code implementing it, shows that the StyleChecker hyperslice provides implementations of check() that perform style checking, and return boolean values to indicate pass or fail. The integration relationship “mergeByName” ensures that these check methods are composed with those from the check() feature, which perform syntax checking. When a check() method is called in the composed hyperslice, therefore, both these check methods will be executed, to check both syntax and style. Each one will return a boolean value to indicate pass or fail of its particular check. What should the overall result be? The best approach in this situation is probably to declare that an expression passes only if it passes both checks. This effect is accomplished with a summary function, whose job is to take an array of results produced

by multiple methods and reduce them to a single result to be returned by the composed method. In this case, the appropriate summary function was coded as demo.StykeChecker.Expression.summarizeCheck, which maps to the composed method DemoSEE.Expression.summarizeCheck. The “set summary function” relationship specifies that this summary function should be used specifically for checks of BinaryOperator objects. In truth, it should be used for all check() methods, but the current release of Hyper/J™ does not permit this to be specified except by listing all the specific cases, and this is the only case that matters in this particular example. The next release of Hyper/J™ is expected to support the following integration relationship:

```
set summary function for operation DemoSEE.check
    to action DemoSEE.Expression.summarizeCheck;
```

5.3.5. Run Hyper/J™

Once the three files described above have been written, it its time to run Hyper/J™ in **Shell Window 1** (see the note in the introduction to this section), using the commands (“java” command all one line; layout here is for ease of reading):

```
java com.ibm.hyperj.hyperj
   -hyperspace   %HYPERJ_DIR%/demo/StyleChecker.hs
   -concerns     %HYPERJ_DIR%/demo/ObjectDimension/concerns.cm
                 %HYPERJ_DIR%/demo/StyleChecker/concerns.cm
   -hypermodules %HYPERJ_DIR%/demo/CheckDisplayStyle.hm
   -verbose
```

Note that we are using the new hyperspace and hypermodule specifications, and two concern mapping files. As before, this will produce the composed hyperslice in the directory DemoSEE, overwriting the prior version of the system (without StyleChecker) we created there before. To keep both composed hyperslices, use a different hypermodule name in the hypermodule specification (demo/CheckDisplayStyle.hm).

5.3.6. Run the Composed Hyperslice

To run the variant of the SEE created above, execute the class files in the composed hyperslice, as before, in **Shell Window 2** (see the note in the introduction to this section):

```
java demo.ObjectDimension.Driver
```

This execution produces output from the Check, Display and StyleChecker features. Comparison with the output of the original program shows absence of the results produced by the Eval feature, and addition of the StyleChecker output. Note that the first expression is now shown to be invalid, because it failed the style check.

5.3.7. Summary

The addition of style checking has demonstrated an important feature of Hyper/J™. As shown earlier, developers need not use Hyper/J™ during initial development—they can use it after 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 derive from the use of new languages or paradigms—the style checker, for example, was written as a standard package in Java™—but, instead, from

the integration (composition) features of Hyper/J™. It has many important advantages and uses, including:

- The ability to treat hyperslice packages as reusable components. When hyperslice packages are used in new contexts, the composition relationships (possibly referring to special-purpose glue code) can include any adaptation that might be necessary (white-box reuse).

- The ability to structure code and design along the same lines as requirements, thereby enhancing traceability, by encapsulating the code that realizes a particular requirement in one or more hyperslice packages [cla99].

5.4. Retrofitting a Design Pattern for Logging

The final change we will explore is the addition of optional logging (or debug tracing) throughout the expression SEE. This modification entails making some or all methods in various classes or features print log messages upon method entry and exit.

Clearly, the logging capability is, conceptually, not specific to the expression SEE—a generic logging capability would make no reference to any expression classes or methods, and could be used in multiple contexts. For this scenario, we assume that such a pre-existing, generic, reusable logging component is available, and can be used to satisfy the new end-user requirement. This particular reusable component uses an implementation of the Observer design pattern, along with a particular instantiation of that pattern to implement logging, as shown in *Figure 4*.[6]

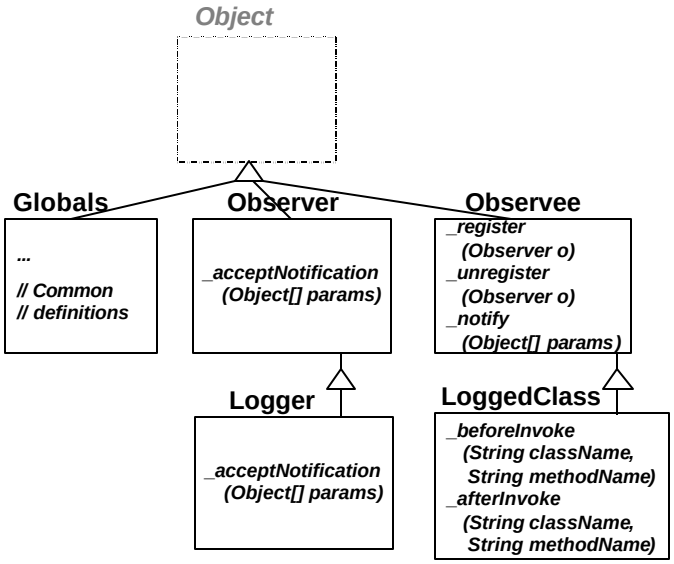

Figure 4. The Logging Hyperslice Package

[6] Other implementations of logging are also possible, of course, and can also be integrated into the expression SEE using Hyper/J. This implementation was chosen here to demonstrate the retrofitting of design patterns into code not originally written with them.

In this case, we will use Hyper/J™ to retrofit the logging capability, which is already encapsulated in a separate hyperslice package (the reusable component), by integrating it into the SEE. Hyper/J™ permits us to make this change additively, as described in the following sections.

5.4.1. Create a Hyperspace Specification File

A new hyperspace specification file, demo\Demo.hs, is needed, to include both the original class files and those making up the StyleChecker and Logging hyperslice package:

```
hyperspace DemoHyperspace
 composable class demo.ObjectDimension.*;
 composable class demo.StyleChecker.*;
 composable class demo.Observer.*
```

5.4.2. Create Additional Concern Mappings

Inclusion of the Observer hyperslice module above automatically specifies new concerns in the ClassFile dimension, but not in the Feature dimension. A simple concern mapping is needed to create a Logging feature concern, and to map everything in the Observer hyperslice to it:

```
package demo.Observer : Feature.Logging
```

This concern mapping is in file demo\Observer\concerns.cm.

5.4.3. Create a Hypermodule Specification File

To instrument methods in the existing code, we define a hypermodule to integrate the Logging feature with any concerns that we want to be logged. For example, the following hypermodule specification file, demo\Demo.hm, creates a version of the expression SEE that contains all the features we have discussed, with all these features logged:

```
hypermodule DemoSEE
   hyperslices:
      Feature.Kernel,
      Feature.Check,
      Feature.Display,
      Feature.Eval,
      Feature.StyleChecker,
      Feature.Logging;
   relationships:
      mergeByName;
      equate operation Feature.Kernel.process,
                       Feature.Check.check_process,
                       Feature.Display.display_process;
      set summary function for action DemoSEE.BinaryOperator.check
          to action DemoSEE.Expression.summarizeCheck;
      bracket "{~_,~<}*" with
         ( before Feature.Logging.LoggedClass._beforeInvoke
                                    ( $ClassName, $OperationName ),
           after Feature.Logging.LoggedClass._afterInvoke
                                    ( $ClassName, $OperationName ),
```

```
            "*" );
    end hypermodule;
```

This hypermodule specification is similar to those we have seen before, except for the "bracket" relationship. This specifies that Feature.Logging.LoggedClass._beforeInvoke() and Feature.Logging.LoggedClass._afterInvoke() are to be used as before/after methods, bracketing other methods as described by the patterns. When they are called, the parameters to be passed to them are not the parameters of the bracketed method, which might be unsuitable, but two strings: the name of the class containing the bracketed method, and the name of the bracketed method itself. The first pattern, "{~_,~<}*", specifies that all methods whose names do not begin with "_" or "<" are to be bracketed. In this example, methods beginning with "_" are excluded because they implement the logging capability itself, and we don't want to log the operation of the logger. Methods beginning with "<" are Java™ constructors, and we choose not to log them either in this example, though doing so is often appropriate. The second pattern, "*", specifies the classes to be logged, in this case all classes. Putting this together, _beforeInvoke() and _afterInvoke() will be called, with the class and operation name parameters, before and after every method in any class whose name does not begin with "_" or "<".

5.4.4. Run Hyper/J™

Once the three files described above have been written, it its time to run Hyper/J™, using **Shell Window 1** (see the note in the introduction to this section), with the commands ("java" command all one line; layout here is for ease of reading):

```
java com.ibm.hyperj.hyperj
   -hyperspace   %HYPERJ_DIR%/demo/Demo.hs
   -concerns     %HYPERJ_DIR%/demo/ObjectDimension/concerns.cm
                 %HYPERJ_DIR%/demo/StyleChecker/concerns.cm
                 %HYPERJ_DIR%/demo/Observer/concerns.cm
   -hypermodules %HYPERJ_DIR%/demo/Demo.hm
   -verbose
```

Note that we are using the new hyperspace and hypermodule specifications, and three concern mapping files. As before, this will produce the composed hyperslice in the directory DemoSEE, overwriting the prior version of the system we created there before. To keep both composed hyperslices, use a different hypermodule name in the hypermodule specification (demo/Demo.hm).

5.4.5. Running Hyper/J with One Control File

In Section 4.1.1.4, we noted that it is possible to specify all of the Hyper/J command line options in a single control file, and to name that file as the first parameter to Hyper/J. This may be considerably more convenient for developers who are running Hyper/J with the same parameters repeatedly, as in the demo scenario presented in this chapter.

We can illustrate the use of a single control file for the scenario presented in Section 5.4.4 above. First, make the following control file, called demo.opt, in your working directory:

```
   -hyperspace   %HYPERJ_DIR%/demo/Demo.hs
   -concerns     %HYPERJ_DIR%/demo/ObjectDimension/concerns.cm
%HYPERJ_DIR%/demo/StyleChecker/concerns.cm
```

```
%HYPERJ_DIR%/demo/Observer/concerns.cm
   -hypermodules %HYPERJ_DIR%/demo/Demo.hm
```

Notes:

- `-concerns` looks like three lines above, but it is really, and must be, a single line.
- You must expand %HYPERJ_DIR% yourself when making this file; Hyper/J currently does not do that for you.
- You can use relative path names if you wish.

Then, in **Shell Window 1**, you may type the following command instead of the one shown in Section 5.4.4:

```
java com.ibm.hyperj.hyperj demo.opt –verbose
```

With this control file, the command to run Hyper/J is clearly much shorter and simpler. In fact, the –verbose option, which was specified on the command line above, could also have appeared in the control file. When control files are used, any Hyper/J option can appear either in the control file or on the command line.

Control files can contain additional information. In the example above, they simply indicated the names of the files in which the hyperspace, concern mapping, and hypermodule specifications could be found. It is possible to include any of these specifications directly in a control file. For example, the following `demo.opt` for the example in Section 5.4.4 is included in the `demo` directory (the blank lines and indentation are for clarity and are not otherwise significant):

```
-hyperspace
    hyperspace DemoHyperspace
    composable class demo.ObjectDimension.*;
    composable class demo.StyleChecker.*;
    composable class demo.Observer.*;

-concerns
    package demo.ObjectDimension : Feature.Kernel

    operation check : Feature.Check
    operation display : Feature.Display
    operation eval : Feature.Eval

    operation check_process : Feature.Check
    operation display_process : Feature.Display
    operation eval_process : Feature.Eval

    operation process : Feature.None

    package demo.StyleChecker : Feature.StyleChecker

    package demo.Observer : Feature.Logging

-hypermodules
    hypermodule DemoSEE
```

```
        hyperslices:
           Feature.Kernel,
           Feature.Check,
           Feature.Display,
           Feature.Eval,
           Feature.StyleChecker,
           Feature.Logging;
        relationships:
           mergeByName;

           equate operation Feature.Kernel.process,
                            Feature.Check.check_process,
                            Feature.Display.display_process,
                            Feature.Eval.eval_process;

           bracket "*"."{~_,~<}*"
                before Feature.Logging.LoggedClass._beforeInvoke
                                       ( $ClassName, $OperationName ),
                after Feature.Logging.LoggedClass._afterInvoke
                                       ( $ClassName, $OperationName );

           set summary function
               for action DemoSEE.BinaryOperator.check
               to action DemoSEE.Expression.summarizeCheck;

    end hypermodule;
```

In this case, it would not be necessary to define separate hyperspace, hypermodule, or concern mapping files; all their contents are directly in this file. The –verbose option, which was shown above on the command line, could equally well have been specified in the options file. Developers are free to use separate files or single control files (or any combination) to best facilitate their particular projects.

To run Hyper/J with this control file, in **Shell Window 1**, you may type the following command instead of the one shown in Section 5.4.4:

```
java com.ibm.hyperj.hyperj %HYPERJ_DIR%/demo/demo.opt –verbose
```

As described in Section 4.2.1.3, it is possible to omit the hyperspace specification if all composable classes are within packages named in the concern mapping. That is the case in this example, so the following shortened control file, `demoshort.opt` (provided in the demo directory), can be used:

```
-concerns
    package demo.ObjectDimension : Feature.Kernel

    operation check : Feature.Check
    operation display : Feature.Display
    operation eval : Feature.Eval

    operation check_process : Feature.Check
    operation display_process : Feature.Display
    operation eval_process : Feature.Eval
```

```
        operation process : Feature.None

        package demo.StyleChecker : Feature.StyleChecker

        package demo.Observer : Feature.Logging

    -hypermodules
        hypermodule DemoSEE
           hyperslices:
              Feature.Kernel,
              Feature.Check,
              Feature.Display,
              Feature.Eval,
              Feature.StyleChecker,
              Feature.Logging;
           relationships:
              mergeByName;

              equate operation Feature.Kernel.process,
                               Feature.Check.check_process,
                               Feature.Display.display_process,
                               Feature.Eval.eval_process;

              bracket "*"."{~_,~<}*"
                   before Feature.Logging.LoggedClass._beforeInvoke
                                          ( $ClassName, $OperationName ),
                   after Feature.Logging.LoggedClass._afterInvoke
                                          ( $ClassName, $OperationName );

              set summary function
                  for action DemoSEE.BinaryOperator.check
                  to action DemoSEE.Expression.summarizeCheck;

        end hypermodule;
```

Note that we could use the simpler –relationships instead of –hypermodules, but then the hypermodule would get the default name, `Composition`. That name would have to be used instead of `DemoSee` in the rules, and the composed output of Hyper/J would go in a directory of that name, affecting the class path for running the composed result.

To run Hyper/J with this control file, in **Shell Window 1**, you may type the following command instead of the one shown in Section 5.4.4:

```
java com.ibm.hyperj.hyperj %HYPERJ_DIR%/demo/demoshort.opt –verbose
```

5.4.6. Run the Composed Hyperslice

To run the variant of the SEE created above, execute the class files in the composed hyperslice, as before, in **Shell Window 2** (see the note in the introduction to this section):

```
java demo.ObjectDimension.Driver
```

This execution produces output from the Check, Display, Eval and StyleChecker features to standard output, and the file expression.log containing the Logging feature output.

5.4.7. Summary

This development scenario entailed the integration of generic, reusable components—the Observer design pattern and logging—into an existing system that had not been designed to use them. This is a common problem for developers, and it occurs in many forms, at all stages of software development—for example, integrating a commercial-off-the-shelf database or library component into software during initial development, or retrofitting a design pattern or other component into the software during the course of evolution. Hyper/J™ facilitates a wide range of such integration activities. The same mechanisms can be used both for integration and customization, as this example shows.

We note that the multi-dimensional approach permits integration and customization using *any* concerns, in any dimensions. Thus, for example, while the developers chose to add logging to a subset of *features*, they could equally well have decided to add it to a subset of *classes*, or to some mix of features and classes. The only difference is in the set of hyperslices specified in the hypermodule. This ability to treat all concerns as equal provides developers the ability to focus their attention on precisely the part of a system that they care about to accomplish their goals.

Chapter

6

Bibliography

[all97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July, 1997.

[and92] E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In O. L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, pages 133–152, Utrecht, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science, no. 615.

[bar96] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[cla99] S. Clarke, W. Harrison, H. Ossher and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 325–339, November, 1999. ACM.

[dso98] D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[gos96] James Gosling, Bill Joy and Guy L. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[har93] 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.

[hol92] I. M. Holland. Specifying reusable components using contracts. In O. L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, pages 287–308, Utrecht, June/July 1992. Springer-Verlag. LNCS 615.

[hyp99] Hyperspace web site, http://www.research.ibm.com/hyperspace.

[jac90] M. Jackson. Some complexities in computer-based systems and their implications for system development. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 344–351, 1990.

[kel99] R. K. Keller, R. Schauer, S. Robitaille and P. Pagé. Pattern-Based Reverse-Engineering of Design Components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 226–235, May, 1999.

[kic97] G. Kiczales. Aspect-oriented programming. *In ECOOP '97: European Conference on Object-Oriented Programming*, 1997. Invited presentation.

[kim99] Doug Kimelman, Multidimensional tree-structured spaces for separation of concerns in software development environments. Position paper, OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99.

[mez98] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October, 1998.

[nus94] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, 20(10):760–773, October, 1994.

[oss88] H. Ossher. A case study in structure specification: A Grid description of scribe. *IEEE Transactions on Software Engineering*, 15(11), 1397–1416, November, 1989.

[oss96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[oss98] H. Ossher and P. Tarr, Operation-level composition: A case in (join) point. In *ECOOP '98 Workshop Reader*, pages 406–409, July 1998. Springer Verlag. LNCS 1543.

[par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[par76] D. L. Parnas, On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, 2(1), March 1976.

[rum98] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[sha96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[tar93] P. L. Tarr and L. A. Clarke. PLEIADES: An object management system for software engineering environments. In *Proceedings of the ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 56—70, December, 1993.

[tar99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, 107–119, May 1999.

[tur98] C. R. Turner, A. Fuggetta, L. Lavazza and A. L. Wolf. Feature Engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design*, 162–164, April, 1998.

[van96] M. VanHilst and D. Notkin. Using roles components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 359–369, October 1996. ACM.

[wol89] A. L. Wolf, L. A. Clarke, and J. C. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.