

Simplified Universe Construction for Hyper/J Composition

Lee Carver
Pnambic Computing
leeca@pnambic.com

Abstract

Hyper/J composition relies on a hyperspace model of the available software units. In the abstract, a hyperspace model provides an unbounded number of dimensions for categorizing software units. Without tool support, these additional dimensions become a tedious bookkeeping problem.

This paper presents a organizational scheme for Hyper/J project and describes a set of tools that rely on this scheme. The organizational scheme allows simple tools to automate much of the bookkeeping effort. The proposed unit annotations extend this scheme to include many crosscutting concerns. In future work, we hope to exploit these annotations for improved composition reports and validation.

1. Introduction

One of the key tasks for a Hyper/J developer is the definition of a universe of composition. Unlike simpler composition tools (e.g. a linker), each composable unit can be associated with multiple properties. Although these properties are organized into a concern hyperspace, it is still a much richer specification than the `LIBPATH` environment variable that is used by many linkers.

With Hyper/J, the demands of hyperspace composition[2] require a universe that carries a rich set of properties. Linkers, and related composition packages, primarily select software units by their name. Hyperspace composition extends this selection process to include an unbounded set of dimensions, each with its own unique coordinates. Without tool support, these additional dimensions become a tedious bookkeeping problem.

This paper presents a hyperspace organizational scheme for Hyper/J projects and describes a universe construction toolkit. The tools automatically generate many of the specification files required for Hyper/J. The overall scheme arose during `FrankenSort` development[1], an experiment in the production use of Hyper/J. A Windows-based version of these tools is available at <http://www.pnambic.com/CPS/HyperJTools/>. The tools exploit widely available Perl, Java, and XML utilities to simplify the implementation.

The analysis tools rely on simple and practical guidelines for organizing Hyper/J development projects. Some Java package directories are augmented with a small `dimension.xml` file. These files define available

dimensions in the universe of composition. These files support automated generation of the Hyper/J specification files.

This rest of the paper follows this outline. The second section presents an overview of Hyper/J and the hyperspace composition model. This section also discusses the motivating example: the construction of an `Application` class from 18 compatible modules. The third section presents the hyperspace organizational scheme and the universe construction toolset that were used to synthesize the `Application` class. The fourth section discusses future work on this tools set. This section proposes several new forms of unit annotation. The fifth section concludes with our overall assessment.

2. Background

This section provides an overview of the bookkeeping challenges for Hyper/J composition. The hyperspace model of unit selection defines a rich *universe of composition*, with many dimensions for categorizing software units. Hyper/J requires multiple specification files to define these dimensions.

Our concrete example is taken from the Application Framework of the `FrankenSort` experiment. This example distributes units from 18 modules across 5 dimensions. Any added modules requires changes to all Hyper/J specification files.

Hyperspaces for Unit Selection

Hyper/J composition relies on a hyperspace model of the available software units. The hyperspace model provides an unbounded number of dimensions for categorizing software units. In well-behaved hyperspaces, no unit can have more than one coordinate (or association) with any single dimension. The hyperspace model allows developers to define a universe of software units with a rich set of properties.

Hyper/J allows a developer to select units for composition based on their coordinates in the hyperspace. With a rich universe of units and dimensions, flexible software construction is straightforward. Individual units can be categorized with many dimensions, such as the programming language, implementation platform, or application feature.

In the `FrankenSort` experiment, I used feature dimensions to select units for composition. These

dimensions required additional declarations to associate the named units with their features dimensions and coordinates. Without tool support, these manual associations become a bookkeeping problem.

With Hyper/J, the Java units for composition are the executable features within the packages: methods, constructors, and fields. Java packages are not software units for Hyper/J. Packages serve primarily as containers of classes and their supporting units.

Hyper/J Universe Definition and Selection

In Hyper/J, two specification files define the universe of composition. The hyperspace specification file enumerates the composable units in the hyperspace universe. The concern-mapping file augments this basic hyperspace with additional dimensions and coordinates. A separate hypermodule specification file defines unit selection and the composition rules.

The hyperspace specification file (`.hs`) identifies each composable class in the universe of composition. This file establishes the Hyper/J's internal [unit-name] dimension. Pattern matching can simplify the inclusion of many classes from the same package. In the **FrankenSort** experiment, every class in the entire package hierarchy is composable.

The concern-mapping file (`.cm`) augments this basic hyperspace with additional dimensions and coordinates. Each class, method, and field can be associated with multiple dimensions. Simple abbreviations propagate class associations to individual members. A developer can also associate individual methods and fields to other dimensions.

The hypermodule specification file (`.hm`) selects units from the universe of composition. Each pair of dimension and coordinate names selects all associated software units. Dependency analysis can select additional software units for any composition. The hypermodule specification file also includes composition relationships for combining individual software units. In the **FrankenSort** experiment, I use only the straightforward `mergebyname` and `order` relationships.

FrankenSort Application Framework

Our motivating example is taken from the **FrankenSort** project. The **FrankenSort** project is an experiment in the use of Hyper/J. This experiment seeks to identify and resolve problems with large-scale software composition. The **FrankenSort** project recreates the behavior of the UNIX sort command using only composition. This paper focuses on the synthesis of the main `Application` class from a large number of compatible modules.

The Application Framework is one of the larger components in **FrankenSort**. It defines common features that are used in many program extensions. Each feature is modularized as a separate Java package. Each package implements its features as members of an `Application` class. The **FrankenSort** Application Framework consists of units from 18 modules distributed across 5 feature-oriented dimensions. These dimensions represent the platform dependent behaviors, execution

stages, argument parsing, option parsing, and help message features.

Within the Application Framework, there is little internal crosscutting. Most interactions are layered, rather than crosscutting. For example, the argument-parsing feature relies only on the execution stages feature. The Application Framework serves primarily as a foundation for future enhancements. Those future enhancements crosscut many of the Application Framework's dimensions. For example, each new option character must extend both the parse options and usage message dimensions.

With 18 modules, the Application Framework component is already large enough that the Hyper/J specification files are awkward to maintain. Each new feature requires changes to all three specification files.

3. Project Organization and Tools

Although Hyper/J takes an egalitarian view of the different dimensions, Java development tools enforce the package hierarchy. Each Java software unit must have a unique implementation in the package hierarchy.

In practice, well-designed modules are strongly associated with a single dimension. In the Application Framework, all units within each Java package are associated with the package's dimension and coordinate. In some packages, individual software units are also associated with additional dimensions.

These practical issues lead to the following hyperspace organizational scheme. This approach exploits the hierarchical organization of many projects, and supports the definition of more flexible composition universes.

Project Organizational Scheme

The basic organization for the software units is a Java package hierarchy. Java packages are used in two roles: as components and as modules. Component packages group modules; module packages implement behavior (i.e. contain code). Only module packages should contain classes and declare software units.

The roles of component packages are more varied. The bottommost level of components collects the module packages into hyperspace dimensions. Higher-level components contain only other components. Subsystem components control interface boundaries. A typical package hierarchy should follow this pattern:

```
package compSubsystem
package compComponent
... other nested Components
package compDimension
    dimension.xml
package modCoordinate
```

In this project organization, all unit implementations are at the lowest level of the package hierarchy (in module packages). All dimension are defined in the component packages that immediately contain the module packages. The `dimension.xml` file that accompanies the Java source files distinguishes these dimension packages.

The Application Framework does not require any nested components. The top five components in the Framework define dimensions, and these components contain only module packages. A subset of the Application Framework structure is shown below.

```

package compApplication
package compPlatform
  dimension.xml
  package modBasic
    class Application
  package modConsole
    class Application
  package modExit
    class Application
  package modProgName
    class Application
package compParseArgs
...
package compParseOpts
...
package compStages
...
package compUsage
...

```

The `compPlatform` component defines a dimension with 5 coordinates. In this implementation, each coordinate (the packages `modBasic`, `modConsole`, `modExit`, and `modProgName`) extends only the `Application` class. Modules in the `compParseArgs` and `compParseOpts` dimension packages also extend other classes, including an `ArgStore` class.

Each `dimension.xml` file identifies a component package that represents a dimension. For each component, this file describes the dimension represented by the package. The `dimension.xml` file for the `compPlatform` component defines a Platform dimension:

```

<?xml version="1.0"?>
<dimension>
  <name>Platform</name>
  <summary>Execution Platform
    support
  </summary>
  <mode>Enhancement</mode>
</dimension>

```

The `<name>` element defines the dimension’s label, and the `<summary>` element provides a descriptive text. These elements are used by the generator applications to define the Hyper/J universe of composition. The `<mode>` element documents the intended interactions for the modules within the dimension. In future work, this element is intended to support validation with the proposed unit annotations.

Hyperspace Analysis Tools

The current hyperspace analysis tools are a simple gather application and three generator applications. The generators create the Hyper/J specification files for routine composition. Future generators and analysis tools may be able to detect or resolve ambiguous or misleading composition specifications.

The gather application walks the source tree and builds a `hyperspace.xml` file that is used for further

processing. The basic content is an XML[3] rendition of the package hierarchy, with the `dimension.xml` contents embedded. The gather application also inspects `.java` source files for unit annotations. Unit annotations are discussed in the Future Work section. Any unit annotations are also added to the resulting `hyperspace.xml`.

At present, generators are available for Hyper/J’s hyperspace specification(`.hs`), concern-mapping(`.cm`), and hypermodule specification(`.hm`) files. These generators use the Saxon XSLT processor to reformat the `hyperspace.xml` content. Future work can provide cross-references or validate composition intent.

The gather application is currently implemented as a Perl-based treewalker application. The three generator applications are largely XSLT macros. This toolset works only for very tightly structure Java code, with one class per file. The current tools do not build or retain ordering information, or other inferable interaction behavior. Despite these limitations, these tools greatly simplified the construction of Hyper/J specification files for **FrankenSort**.

4. Future Work

The hyperspace organizational scheme is a simple extension of the Java class hierarchy. As such, it can only capture hierarchical concerns and relationships. The organizational scheme’s package-level `dimension.xml` is inadequate for crosscutting concerns. Non-hierarchical relationships require the declaration of addition properties. Fine-grained composition requires dimensions and coordinates for individual methods and fields.

The proposed unit annotations allow the declaration of wide-ranging, non-hierarchical relationships. Currently, the universe construction toolkit collects unit annotations, but does not process them. In the future, the toolkit is expected to use the annotations to analyze and validate the *universe of composition*.

Unit Annotations

Unit annotations are stylized Java comments that can be easily accessed by the gather application. Each unit annotation defines an XML element that describes its associated Java unit. Two kinds of XML elements are captured by unit annotations. Interaction elements – such as `<declare>`, `<extend>`, and `<use>` – define the permissible interactions among composable units. The `<concern>` element specifies additional dimensions of concern for each unit.

Unit annotations take the form of a Java comment with a distinctive XML-like prefix (“//<”). The following character, a “gather marker”, indicates the lexical handling for this annotation. A “=” gather marker indicates that the XML element is closed after the first non-annotation line. A “+” gather marker indicates that the XML elements should remain open. A “/” gather marker forces the XML element closed.

The three basic interaction elements – `<declare>`, `<extend>`, and `<use>` – define the anticipated role for each software unit. During composition, a named

interface should not be extended or used unless there is exactly one declaration. In addition, validation could confirm that each interface is extended or used only by compatible units. Additional interaction elements describe combine roles for software units. For example, the `<define>` element indicates an interface that is both `<declare>d` and `<extend>ed`.

The interaction elements use the name and form attributes to define the symbolic name and unit form of each software unit. The name property defines the name and type signature for each unit. In Java applications, the four widely used unit forms are `class`, `method`, `field`, and `ctor`. The `mode` attribute is only used with `<declare>` and `<extend>` elements. This attribute defines the intended composition behavior for the units. The two basic composition modes are `fusion` and `dispatch`. Some extended and aggregate composition modes are also supported (e.g. `singleton`, `after`, `before`).

Unit annotations from the `modApplication` module show a typical set of definition elements. (In Java code, these annotations easily fit on one line.)

```
//<+ define name="Application"
        form="class"
        mode="fusion" >
//<= define name="execute()"
        form="method"
        mode="fusion" >
//<= define name="main()"
        form="method"
        mode="singleton" >
//</ define >
```

Typical extending annotations are:

```
//<= extend name="execute()"
        form="method"
        mode="ordered-unit" >
//<= extend name="execute()"
        form="method"
        mode="ordered-unit" >
```

In order to ensure referential completeness in extension modules, interface uses are also annotated. Typical use annotations are:

```
//<= use name="asrgStore"
        form="field" >
//<= use name="execute()"
        form="method" >
```

Additional dimensions of concern are defined with the concern annotations. Concern annotations are placed between interactions elements and the defining Java declaration. Typical concern annotations take the form :

```
//<= concern dimension="componentName"
        coordinate="moduleName" />
```

All of these annotations are highly experimental. For robust program composition, both component name and module name will need flexible namespace management mechanisms. Several of these annotations are expected to evolve with more experience.

5. Conclusions

The hyperspace organizational scheme and the universe construction toolkit provide a proof-of-concept implementation for Hyper/J assistance. This project structure and these tools demonstrate that simple structuring rules for project organization can greatly contribute to a simplified specification of the composition. I look forward to extending these tools to support a rich set of validation and reporting behaviors.

The tools have been very useful in the construction of the **FrankenSort** Application Framework. The addition of latter modules, especially the `modExit`, `modProgName`, and `compUsage` packages required minimal manual intervention. Manual changes were limited to the re-insertion of the un-constructed order information.

References

- [1] L. Carver, *Building a Real-World Application with Aspect-Oriented Modules and Hyper/J*, Master's thesis, University of California, San Diego, Department of Computer Science and Engineering, June 2002
- [2] IBM Corporation, *Hyper/J User and Installation Manual*, 2000.
- [3] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, Second Edition, 6 October 2000.