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 standardized scheme for Hyper/J project organization and describes the related software tools. This approach allows simple tools to automate much of the bookkeeping effort. Most software units are organized into implementation modules, and small `dimension.xml` files describe the overall hyperspace structure.

The current hyperspace analysis tools are a simple `gather` application and a number of generator applications. The generators create the Hyper/J input files for routine composition. The generated inputs simplify the process of extending the hyperspace of software units.

Unit annotations augment the package-level information currently used by the generator applications. 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 user is the specification of the universe of composition. This task is common to many software construction tools. Linkers rely on the LIBPATH environment variable; the Java SDK standard relies on the ClassPath variable.

With Hyper/J, the demands of hyperspace composition require a universe that carries a rich set of properties. Standard composition packages, like linkers, select software units by 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 standardized scheme for Hyper/J project organization and describes the related software tools. The tools automatically generate many of the specification files required for input to Hyper/J. The overall scheme arose during the **`FrankenSort`** project[me], an experiment in the production use of Hyper/J. A Windows & ActivePerl version of these tools is available at http://www.pnambic.com/CPS/HyperJTools/. The tools exploit widely available XML utilities to simplify the implementation.

The analysis tools rely on simple and practical guidelines for organizing Hyper/J development project. Some Java `package` directories are augmented with a small `dimension.xml` file. These files define the available dimensions in the project tree, and automate generation of simple Hyper/J universe definitions.

This paper is laid out as follows. 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 development standards and analysis tools that were used to synthesize the `Application` class. The fouth section discusses future work on this tools set. Unit annotations are a . The fifth section concludes with our overall assessment. Sample output from the toolset is shown in the Appendix.

2. Background

Software composition involves a selection of software units from a universe of available units. In conventional practice, this universe of available units is specified obliquely, though modules that bundle units and a LIBPATH (or ClassPath) environment variable that bundle modules.

In this arrangement, the essential property for each unit is its name. Units are selected for composition (or execution) primarily by their name.

Hyperspaces and Hyper/J 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 then one coordinate (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 selects units for composition based on their coordinates in the hyperspace. With a rich universe of units and dimensions, flexible software construction is straightforward. However, specifying the universe with all its units and associations is a tedious bookkeeping challenge.

Hyper/J automates the construction of its own internal [unit-name] dimension for software units. This dimension associates each software unit with its hierarchical name in the package hierarchy. Although this dimension is "essential" for the composition process, it is often irrelevant for unit selection. In **`FrankenSort`**, I used feature dimensions to select units for composition.

Feature dimensions require manual definition of additional dimensions and associations. Hyper/J allows each software unit to be manually associated with other properties (or coordinates) on many independent dimensions. Without tool support, these manual associations present a tedious bookkeeping problem.

In Hyper/J, Java packages are not software units. The Java units for composition are the executable features within the packages: methods, constructors, and fields (which have implicit `get()` and `set()` methods). Packages serve primarily as containers of classes and their supporting methods.

Hyper/J Universe Definition and Selection

Although Hyper/J automates the construction of a unit-name dimensions, user defined dimensions require auxiliary specification files. In Hyper/J, the universe of composition and the selected composition are specified through a combination of input files.

The hyperspace specification file (`.hs`) enumerates the composable units in the hyperspace universe. This file identifies each composable class in the universe of composition. In our example, every class in the entire package hierarchy is composable. This file establishes the Hyper/J's internal [unit-name] dimension.

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 units. Each pair of dimension and coordinate names selects all associated software units. Dependency analysis can select additional software units for composition. The hypermodule specification file also includes composition relationships for combining individual software units. In our example, we 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. These features are implemented as methods in an `Application` class. The Application Framework consists of 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. Although argument parsing relies on execution stages, this is a simple interaction. The 5 dimensions define largely orthogonal features. The Framework serves as a foundation for future enhancements. Those future enhancements crosscut many of the Application Framework's dimensions. For example, each added option must extend both the parse options and usage message dimensions.

At 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 concerns suggest a project organization similar to the scheme below. This approach exploits the hierarchical organization of many modules, and supports the definition of more flexible composition universes.

Project Organization

The basic organization for the software units is a Java package hierarchy. Java packages are used in two roles: as components and as modules. Components group modules; modules implement code. Only module packages should contain classes and define 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 comp{Subsystem}
  package comp{Component}
    …
    package comp{Dimension}
      dimension.xml
      package mod{Label}
```

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 internal nested components. The top five components in the Framework define dimensions, and 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`, `modeExit`, and `modProgName`) extends only the `Application` class. Modules in the `compParseArgs` and `compParseOpts` dimensions also extend an `ArgStore` class.

A `dimension.xml` file identifies each 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 the Platform dimensions:

```
<?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 are used in generated outputs to define Hyper/J dimension. The `<mode>` element documents the intended interactions for the modules within the dimension. In future work, this element supports validation with more detailed annotations.

Hyperspace Analysis Tools

The current hyperspace analysis tools are a simple `gather` application and a number of generator applications. The generators create the Hyper/J input files for routine composition. Output from these generators is shown in the appendix. 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 `Composite.xml` file that is used for further processing. The basic content is an XML rendition of the package hierarchy, with the `dimension.xml` contents embedded. It also inspects the `.java` source files for unit annotations. Unit annotations are discussed in the Future Work section. These unit annotations are also added to the `composite.xml` result.

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 `composite.xml` content for Hyper/J. Future work can provide crossreferences or validate composition intent. Several possible extensions are discussed in the Future Work section.

4. Future Work

The current generator tools work fine for simple, orthogonal dimensions. However, these tools do not address more complex domains with multiple, intersecting dimensions. Simple, package level dimension binding is inadequate. Fine-grained composition requires dimensions and coordinates for individual methods and fields.

The proposed unit annotations provide a mechanism for defining the additional dimensions and coordinates for fine-grained composition. We implement annotations these as special comments in the Java source. Enhanced toolsets should capture and process this additional information.

The current `gather` application is a straightforward Perl 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 simplify the construction of common Hyper/J specification files.

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 the associated software 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 each unit. During composition, a named interface should not be extended or used unless there is exactly one definition. In addition, validation could confirm that each interface is extended or used only by compatible units. Other interaction elements describe combination roles. 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 form of each software unit. Four forms widely used in Java applications are `class`, `method`, `field`, and `ctor`. The `mode` attribute is only used with `<declare>` and `<extend>` elements. This attributes 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. `singelton`, `after`, `before`).

The unit annotations from the modApplication module shows a typical set of definition elements.

```
//<+ 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 completeness in extension modules, interface uses are also annotated. Typical annotations are:

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

We would like to add direct support for introduction of alternatives. A dispatch mode with the following structure could support the introduction of alternatives. In **FrankenSort**, limitations in Hyper/J forced us to use surrogate sites for each alternative. This eliminated the need for explicit `dispatch` annotations.

```
  //<= define name="onOpt()" form="method" mode="dispatch" …
             select="switch(ch)" >
  //<= extend name="onArg()" form="method" mode="dispatch" …
             selector="case 'M':" >
```

Universe definition are added between the declaring annotation and the declaring Java text.

```
  //<= concern dimension="{componentName}" label="{moduleName}" />
```

These annotations are largely experimental. For robust program composition, both component name and module name will need flexible namespace management mechanisms. XML namespaces provide a powerful mechanism, and may be sufficient for integrating independently developed subsystems.

5. Conclusions

The project organization and hyperspace analysis tools provide a proof-of-concept implementation for Hyper/J assistance. They demonstrate that simple structuring rules for project organization contributes to a simplified specification of the composition. We look forward to extending these tools to support a rich set of validation and reporting features.

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 merely the re-insertion of the un-constructed order information.

References

[1] The AspectJ Team, *The AspectJ Programming Guide*, version 1.0candidate 1, October 2001.

[2] L. Carver and W. G. Griswold, Sorting out Concerns, First Workshop on Multi-Dimensional Separations of Concerns, OOPSLA'99, November 1999.

[3] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong, Structuring Operating System Aspects, Communications of the ACM, vol. 44, no. 10, October 2001.

[4] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[5] Free Software Foundation, `textutils-1.22`, January 1997.

[6] IBM Corporation, *Hyper/J User and Installation Manual*, 2000.

[7] IEEE, *Standard for Information Technology – Portable Operating System Interface (POSIX) Part 2 – Shell and Utilities*, IEEE Standard 1003.2-1992, 1992.

[8] D. L. Parnas, On the Criteria to be Used for in Decomposing Systems into Modules, pp 1053-1058, *Communications of the ACM*, vol. 25, no. 12, 1972.

[9] D. L. Parnas, Designing Software for Ease of Extension and Contraction, pp 301-307, *IEEE Trans. On Software Engineering*, vol. SE-2, no. 4, Dec. 1976.

[10] A. Robbins, *Unix in a Nutshell*, O'Reilly, 1999.

[11] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, *Proceedings of the International Conference on Software Engineering (ICSE'99)*, Los Angeles, 1999.

6. Appendix

This section shows the actual output from the hyperspace analysis tools

This is the generated hyperspace specification file for the Application Framework subsystem.

This is the generated concern-mapping file for the Application Framework subsystem

```
// From E:/Users/LeeCa/Research/FrankenSort/Source
hyperspace FrankenSort
    // Include Argument Parsing Package
    // Coordinate ParseArgs.modBasic
    composable class compApplication.compParseArgs.modBasic.Application;
    composable class compApplication.compParseArgs.modBasic.ArgParsingException;
    composable class compApplication.compParseArgs.modBasic.ArgStore;
    // Coordinate ParseArgs.modOnArgDispatch
    composable class compApplication.compParseArgs.modOnArgDispatch.Application;
    composable class
compApplication.compParseArgs.modOnArgDispatch.OnArgDispatch;
    composable class compApplication.compParseArgs.modOnArgDispatch.OnArgHandler;
    // Coordinate ParseArgs.modStdOnArgDispatch
    composable class
compApplication.compParseArgs.modStdOnArgDispatch.Application;
    composable class
compApplication.compParseArgs.modStdOnArgDispatch.StdOnArgDispatch;

    // Include Option Parsing Package
    // Coordinate ParseOpts.modAdvancedOnOptDispatch
    composable class
compApplication.compParseOpts.modAdvancedOnOptDispatch.AdvancedOnOptDispatch;
    composable class
compApplication.compParseOpts.modAdvancedOnOptDispatch.Application;
    // Coordinate ParseOpts.modBasic
    composable class compApplication.compParseOpts.modBasic.Application;
    composable class compApplication.compParseOpts.modBasic.OptOnArgHandler;
    composable class compApplication.compParseOpts.modBasic.OptParsingException;
    composable class compApplication.compParseOpts.modBasic.OptStore;
    // Coordinate ParseOpts.modNaiveOnOptDispatch
    composable class
compApplication.compParseOpts.modNaiveOnOptDispatch.Application;
    composable class
compApplication.compParseOpts.modNaiveOnOptDispatch.NaiveOnOptDispatch;
    // Coordinate ParseOpts.modOnOptDispatch
    composable class compApplication.compParseOpts.modOnOptDispatch.Application;
    composable class
compApplication.compParseOpts.modOnOptDispatch.OnOptDispatch;
    composable class compApplication.compParseOpts.modOnOptDispatch.OnOptHandler;
    composable class
compApplication.compParseOpts.modOnOptDispatch.OptDefinitionException;
    composable class compApplication.compParseOpts.modOnOptDispatch.OptIgnore;
    composable class
compApplication.compParseOpts.modOnOptDispatch.OptUnrecognized;
```

```
        // Include Execution Platform support
        // Coordinate Platform.modBasic
        composable class compApplication.compPlatform.modBasic.Application;
        // Coordinate Platform.modConsole
        composable class compApplication.compPlatform.modConsole.Application;
        // Coordinate Platform.modExit
        composable class compApplication.compPlatform.modExit.Application;
        // Coordinate Platform.modProgName
        composable class compApplication.compPlatform.modProgName.Application;

        // Include Application Stages support
        // Coordinate ApplicationStages.modEvaluate
        composable class compApplication.compStages.modEvaluate.Application;
        // Coordinate ApplicationStages.modInitialize
        composable class compApplication.compStages.modInitialize.Application;
        // Coordinate ApplicationStages.modParseArgs
        composable class compApplication.compStages.modParseArgs.Application;
        // Coordinate ApplicationStages.modRegisterExtension
        composable class compApplication.compStages.modRegisterExtension.Application;
        // Coordinate ApplicationStages.modTerminate
        composable class compApplication.compStages.modTerminate.Application;

        // Include Usage Message support
        // Coordinate UsageMsg.modBasic
        composable class compApplication.compUsage.modBasic.Application;
        // Coordinate UsageMsg.modOptDescr
        composable class compApplication.compUsage.modOptDescr.Application;
        composable class compApplication.compUsage.modOptDescr.OptDescr;
        composable class compApplication.compUsage.modOptDescr.OptDescrStages;
        composable class compApplication.compUsage.modOptDescr.StdOptDescr;
```

This is the generated hypermodule specification file for the Application Framework

```
// From F:.Users.LeeCa.Research.FrankenSort.Source.
hypermodule Composite
    hyperslices:

        // Include Argument Parsing Package
        ParseArgs.modBasic,
        ParseArgs.modOnArgDispatch,
        ParseArgs.modStdOnArgDispatch,

        // Include Option Parsing Package
        ParseOpts.modAdvancedOnOptDispatch,
        ParseOpts.modBasic,
        ParseOpts.modNaiveOnOptDispatch,
        ParseOpts.modOnOptDispatch,

        // Include Execution Platform support
        Platform.modBasic,
        Platform.modConsole,
        Platform.modExit,
        Platform.modProgName,

        // Include Application Stages support
        ApplicationStages.modEvaluate,
        ApplicationStages.modInitialize,
        ApplicationStages.modParseArgs,
        ApplicationStages.modRegisterExtension,
        ApplicationStages.modTerminate,

        // Include Usage Message support
        UsageMsg.modBasic,
        UsageMsg.modOptDescr,
               ;
    relationships:
        mergeByName;
    end hypermodule;
```

As generated, this file requires that a developer remove the last comma, following the `Usage.modOptDescr` hypersplice reference. In practice, other additions for `order` dependencies are also needed before a successful Hyper/J composition.