Desarrollo de Software Orientado a Aspectos, DSOA 2006
Asociado a XV Jornadas de Ingeniería del Software y Bases de Datos
J. Araújo, J. Hernández, E. Navarro y M. Pinto (Eds)
Sitges (Barcelona), Octubre - 2006

HYPER/NET: MDSOC SUPPORT FOR .NET

Tiago D. Dias[1a], Ana M. Moreira[1b]

1: Departamento de Engenharia Informática
Faculdade de Ciencias e Tecnologia
Universidade Nova de Lisboa
Monte de Caparica
2829-516 Caparica
a: e-mail: tdias@ptsoft.net, web: http://www.ptsoft.net/tdd
b: e-mail: amm@fct.unl.pt, web: http://www-ctp.di.fct.unl.pt/~amm/

Keywords: Multidimensional separation of concerns, partial types, .NET, multidimensional unit testing, composition.

Abstract. *This article presents Hyper/Net, a tool we developed to provide multidimensional separation of concerns (MDSOC) support in .NET. MDSOC is an approach to analysis, design and code artifact modularization; Hyper/Net addresses the modularization of code. We introduce our preliminary MDSOC implementation method using .NET 2.0 Partial Types. Hyper/Net further extends this method, for any .NET version, by supporting the composition of methods, in a way similar to Hyper/J. Additionally a multidimensional approach for unit testing is presented.*

1. INTRODUCTION

Multidimensional separation of concerns (MDSOC) proposes the decomposition of software in multiple dimensions, broken down into concerns, eventually spanning different software paradigms [1]. This approach is thus more general and potentially powerful than Aspect Oriented Programming [7] in its' usual two dimensional form (objects extended by aspects).
By supporting the MDSOC approach in .NET languages (C#, VB.Net) the expected benefits are:

1. to provide a closer mapping between requirements and code organization, which is difficult with plain object-oriented programming languages;
2. to enable programmers to work on different concerns of the same object/unit separately (in space and/or time) bringing these together natively (instead of mixing separately modified source code files);

3. to support manageable evolution without the need for refactoring;
4. to offer the creation of product families through mixing-and-matching.

This paper focuses on benefits listed 2. and 3. Section 2 introduces some Basic MDSOC concepts. Section 3 presents a simple method to achieve very basic MDSOC with .NET's partial types. Section 4 discusses a classic example of MDSOC based software development using only partial types. Section 5 introduces Hyper/Net, our tool, which extends partial types for further MDSOC support. Section 6 exemplifies further implementations to the example from Section 4. Section 7 presents some related work, leading way to the conclusions in Section 8.

2. AN OVERVIEW TO MDSOC

In MDSOC, concerns are generic grouping elements [1,2]. Each concern in MDSOC exists in the context of a dimension. Dimensions are more general aggregators. The number of dimensions is virtually limitless and can embody different kinds of artifacts, namely different programming languages. Concerns discretely populate a dimension in such a way that no two concerns overlap in the same dimension. That is, a specific dimension doesn't have any cross-cutting concerns. Overlapping concerns may exist between dimensions, but such crosscutting is already sliced at the dimension level. Concerns are composed of artifact instantiations or units, such as objects, functions or procedures. Units can span several concerns in different dimensions but the same object never appears in more than one concern in the same dimension.

The space defined by the dimensions identified is referred to as the hyperspace [3] or, sometimes, concern space. In order for units to implement a specific concern in a given dimension, units regarding that concern must be combined in an independent module, this is called a hyperslice. As units can be crosscutting, it is necessary to decompose each of these units and only include in the hyperslice the component that is related with the hyperslice concern. Unit decomposition is done during hyperspace definition. This can either be done on existing software or be part of the development process itself.

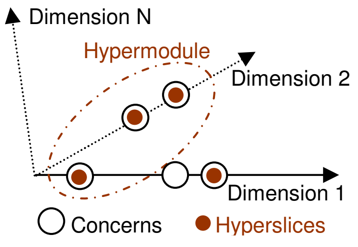

Figure 1. An hyperspace representation.

Hyperslices usually offer very specific functionality and are of limited standalone usage. To regain power, hyperslices are composed into hypermodules by using rules on how the components of each hyperslice integrate with each other. These are called composition rules. Hypermodules can crosscut several dimensions, as can be seen in Figure 1, and can also be composed with and on top of each other to attain complete software systems.

3. MDSOC WITH PARTIAL TYPES

As part of new language features Microsoft introduced partial types with the C# and VB.Net 2.0 language definitions. Partial types use a class modifier construct (`partial`) that enables separating class definitions throughout as many files as desired. Figures 2 and 3 exemplify the usage of partial types to implement different methods for the same class in separate files.

```
partial class Fish {
    public void Eat(IEdible food) { ... }
}
```

Figure 2. Declaration of the partial class Fish in File1.

```
partial class Fish {
    public void Sleep(int minutes) { ... }
}
```

Figure 3. Declaration of the partial class Fish in File2.

Here we explore the usability of partial types to organize code by adopting the multidimensional separation of concerns (MDSOC) approach. To achieve this we need to: (1) define hyperspaces; (2) define composition.

Throughout this paper we will consider hyperspace units at the class/object level. To define a hyperspace we need to determine dimensions and concerns; note that these can be extended afterwards. To populate the dimensions, existing units must be decomposed or new units must be created. (These units span several concerns.) We achieve unit decomposition by separating class declarations in different files, each containing a partial class definition. To organize our hyperspace we use a very simple approach: having defined our dimensions and the concerns in each, we create a directory structure where each dimension has a root directory and inside is one directory per concern. Files pertaining to a concern are simply placed inside the concern directory. Additionally, we can add comments or attributes to the code in order to identify the concern. We cannot use namespaces to map each partial class implementation to a concern, as Hyper/J can, since partial types require each partial class declaration to be in the same namespace.

The second step, define composition, is automatically provided in our approach, being supported by the compiler itself. Partial classes are brought together into a single piece which holds the entire class implementation in the compiled code. Composition is done by class name and doesn't support any merge, override or other advanced composition mechanisms. Support for such mechanisms is added by our tool (Hyper/Net) and is described in Section 4.

4. CASE STUDY: THE EXPRESSION SEE

As an experiment of the proposed method we implemented the example used in [1, 3, 4], which is also used in the Hyper/J demo [5]. The expression software engineering environment (SEE) is a simple object-oriented (OOP) implementation of mathematical expressions with numbers, operators (+, - and assignment) and variables. It supports features as printing expressions, evaluating an expression value and checking the syntax of an expression. For this

example, an hyperspace with a single dimension, divided into concerns, will be defined.
First of we create a directory for a Kernel concern, the kernel concern is a base concern where we declare each class and offer basic functionality: constructors, related private variables and optionally get/set methods (we have none of these type). We use the Kernel concern to organize our class hierarchy in the following way:

- An Expression is an abstract superclass for all the other classes.
- Binary operators share commonalities captured in a class of their own (BinaryOperator) which derives from Expression. Subclasses of this are Plus, Minus and Assignment.
- Number and Variable extend Expression with the expected functionality.
- There is an additional Test class implementing unit tests for each concern. Namely testing object construction for the Kernel concern.

All the classes declared in the Kernel directory (concern) are defined as partial classes so we can further enrich them for other concerns. A concern Display focuses on printing expressions on the screen. Therefore, in another directory of the Feature dimension (root directory) we have the previous classes with only the implementation of a `Display()` method.
The display hyperslice also contains the definition of unit tests regarding the display method. By placing separate unit tests in each hyperslice we are optimizing the system for mix-and-match operations[1], retaining the full test driven development capabilities. Such mix-and-match operations can easily be done by adding the directory of a concern as part of the project for compilation, or removing it. For example, we can easily produce a version of the Expression software without display capabilities by removing the Display directory (one click functionality in .NET IDEs[2]) and compiling a new version of the project. Note that unit tests for the feature remove are also removed; if Testing was implemented as a concern, we would have to manually remove them for the removed concern so that we could compile the project.
The Evaluation and Check concerns are implemented similarly to Display concern, each one adding a new method to the classes (`Eval()` and `Check()`, respectively). Not every class has a partial implementation in each concern. For example, the check functionality of binary operators is implemented in the parent class and inherited by all subclasses.
[1] proposes an extension to the Expression SEE that consists in adding a Style checking concern. This new concern should offer its' functionality through the `Check()` method introduced in the Check concern. This would enable existing code that uses expression checking to do style checking without needing to be changed. Here we find a major limitation in our initial model: if we declare another partial class for any of the implemented classes offering another implementation for the Check method, the compiler will detect a syntax error as the Check method is defined twice. (Remember that each partial class is composed in a unique class additively for all elements declared in the partial declarations.)

[1] Namely removing concerns and adding new ones as required, to produce different flavors of the Expression software product.

[2] Microsoft's Visual Studio is the most common IDE for the .NET platform, but for this project we used an open source IDE: SharpDevelop, available at http://www.sharpdevelop.net/.

An alternative would be to create a new method that explicitly combined both functionalities of the different checks. This method would have to replace the existing syntactic check calls in software that required using both checks. The new method would reference both kinds of checks explicitly and thus be dependent on both. Hyper/J attains the desired transparent effect by introducing elegant composition strategies that we have also implemented in our Hyper/Net prototype described in the next section.

5. EXTENSIONS FOR FULL MDSOC SUPPORT

Our initial approach only enables the composition of classes in a very specific way; still it can prove valuable, especially as no additional software is required to apply such an approach. Hyper/Net builds on this initial work by providing explicit composition constructs that elevate methods[3] to MDSOC units, instead of having only classes working as units.

At this point Hyper/Net's composition constructs take the form of .NET attributes. These can be applied to any program element (class, interface, variable, etc.), but we only take into account composition attributes for methods. Hyper/Net supports three composition constructs found in Hyper/J: `override`, `merge` and `bracket`. In Hyper/J, the first two constructs define how elements are composed. Override replaces a set of other methods and merge captures the functionality of each individual instance in a new 'super-method'. Bracket enables inserting actions occurring before and after a specific method; this construct is similar to AspectJ's [8] `after` and `before` advices and was already present in Hyper/J.

Overrides is the simplest composition construct (see Figure 4); with this, only one of the conflicting methods can define this attribute and the remaining methods are simply removed.

```
[MethodMerge(MethodMergeAction.Override)]
```

Figure 4. Syntax of the override composition attribute.

Both overrides and merge compositions are defined using the `MethodMerge` attribute; that has to do with the common implementation for both composition methods, and should be corrected in a non prototypical version.

Merge composition embodies existing method functionality as a 'super-method' that replaces the previous ones. This is done by invoking each instance of the existing methods, as illustrated in Figure 5. At least one conflicting method has to define a merge attribute for this operation.

```
[MethodMerge(MethodMergeAction.Merge, <Priority>, <MergeResultMethod>)]
```

Figure 5. Syntax of the merge composition attribute.

In merge compositions we additionally capture the functionality of the `order` composition

[3] The Hyper/Net prototype only supports the composition of methods (other than constructors). We plan to extend support for all other types of elements at the class level (variables, properties, etc.).

rule in Hyper/J with the priority argument (which is optional[4]). The priority argument defines a total order in which merged methods will be executed; it's our intent to offer a relative ordering feature in the future. We can also define a method that will merge the results of each original method invocation (see Figure 6). Such method must be local in the class and receives a list of result objects to return only one object of the same type[5].

```
public delegate object MethodMergeResult (params object[] mergedResults);
```

Figure 6. The method result merger is a delegate handler.

The bracket constructor (see Figure 7) enables preceding the invocation of a method with another method (before method) that receives its' arguments and information about the original method. It also enables (and requires in the current Hyper/Net implementation) following methods with the invocation of another method. The method invoked after the initial invocation (after method) receives the same information as the before method along with the return result of the intercepted method. The invoked methods must be local to the class and implement the delegates depicted in Figure 8.

```
[MethodBracket(<Before method>, <After method>)]
```

Figure 7. Syntax of the bracket composition attribute.

```
public delegate void BeforeMethod(MethodBase method, params object[]
 parameters);
public delegate object AfterMethod(MethodBase method, object returnValue,
 params object[] parameters);
```

Figure 8. Before and after methods are delegate handlers.

Contrary to Hyper/J, we do not define default merging actions as we rely on the fixed merging of partial types. We think this might prove simpler and more intuitive to use for the programmer.

Hyper/Net works as a pre-compilation tool that processes source code. Both C# and VB.Net are supported at this point. Hyper/Net receives as input a project file and reads the source code used for compilation of the project[6]. The source-code is pre-processed in order to merge all the files into one. This is done by moving all using/import directives to the beginning of the code. Then Hyper/Net uses the parser implemented by NRefactory[7] to produce an AST (abstract syntax tree) for the code.

[4] When the priority is not defined, methods will have a default priority of -1.

[5] Type checking for these methods is done only at runtime; it should be done as part of Hyper/Net.

[6] MSBuild project files are like buildfiles, written in XML, containing, among much more, information regarding the source-code files that need to be compiled to produce binary output.

[7] NRefactory is, at the time of writing, an undocumented project from the SharpDevelop development team.

Following the initial processing stages there is a composition preparation stage. It is in this stage that partial types are merged into a single type declaration, but first there is a similar merge that is done with namespaces; this brings all common namespace entries below the same namespace node. Partial types are fully merged: merging inheritance and interface implementations, attribute declarations and, of course, all of the scattered[8] element declarations (this is purely additive). The composition is done in the next step.
The final stages of Hyper/Net processing are the core of our work. Merge and override composition is done in the same step. The AST is visited once again, this time searching for repeated instances of the same method; method signatures (name, return and argument types) are taken into consideration to determine different instances of the same method. Remember that scattered methods have already been brought under the same tree elements in the AST in the previous steps. At this point, matching methods are searched for attributes in order to determine the correct course of action. If one (and only one) of the methods has an override attribute, the remaining methods are removed. Otherwise, the methods must define merge composition. Merge composition consists in renaming the existing methods and creating a new ‘super-method’ that will call each one of the previous. The priorities defined are used for ordering the invocation of each method. The result of each invocation is kept in a local variable. At the end of the invocation process an optional result merging method is invoked. This method, which must be defined as a local class method, takes as arguments a list of results and returns only one, which, in turn, is returned by the ‘super-method’.
Bracketing searches directly for bracket attributes in methods. Once found, a statement that gets the original method meta-information (populates a System.Reflection.MethodBase object) is injected at the beginning of the original method. Next, the before method is invoked receiving the method meta-information and the original method arguments. Finally, the existing return statement is replaced by the population of a result variable using the returned expression, and the after method is injected replacing the existing return and receiving the original result along with the same arguments provided to the before method.
Finally, Hyper/Net outputs the composed code in the language of choice (either C# or VB.Net), as a single source file that can then be compiled.
Hyper/Net has been implemented itself using the partial type MDSOC approach presented earlier. Furthermore, Hyper/Net doesn’t require a 2.0 compiler because the partial types defined in source code are processed internally after parsing in order to facilitate the composition phases. As a result, even though partial types are used in Hyper/Net’s MDSOC source code, the compiler used with the resulting code needs not be aware of these partial classes and thus can be a version 1.x compiler.
Hyper/Net supports multiple compositions for each element as attributes are kept and propagated from composition to composition, namely automatically introduced methods retain the attributes of their originators.

[8] Please note that these are scattered only from the object dimension point of view, thus our intention to discard such dimension in our analysis.

6. REVISITING THE EXPRESSION SEE

We'll now get back to the Expression SEE example in order to put Hyper/Net to work at extending the existing features.

Getting back to the style check feature, that we were unable to introduce with only partial types, we can now introduce it easily. We simply created a new concern/hyperslice (StyleCheck) in which we provided partial implementations for the `Check()` method. As a simplification to the original feature introduced in a demo with Hyper/J, our style check simply checks that the name of variable elements is smaller than 5 characters. This way our concern only required a default implementation for expression that returns true; an implementation for the binary operator makes sure each side expression is correct; finally, the implementation for the variable object checks the length.

Each of these methods has a `MethodMerge` attribute declaration applied to it. Figure 9 presents the merge attribute for the `Check()` method in the style check hyperslice. The attribute action used is `Merge`, defining that the method will be merged with all other methods with the same signature. We used a priority level of -10, which is inferior to the default merge priority (-1), so this method is invoked after the syntactic check. The method in charge of creating the return value receives the two results and returns true if both are true.

```
[MethodMerge(MethodMergeAction.Merge, -10, "mergeCheckResult")]
```

Figure 9. Attribute declaration for merging the check feature.

Going back to the syntax check feature in Section 5 we can recall that in this concern, the method `Check()` was additionally implemented for the assignment and number classes. We haven't defined explicitly a check method for these classes in the style check concern; instead, we rely on inherited functionality from the expression class. When partial classes are brought together there is a side effect of the fixed merging of partial types; the methods defined in the syntactic check concern for the number and assignment classes override the default merged implementation for the expression class. This requires us to declare explicitly the default functionality for both classes in the style check concern and the merging attribute. This should be unnecessary and the continuation of this research should provide an elegant solution for this problem.

During the implementation of the new features we used unit testing to validate the correctness of the resulting code. Our approach to unit testing in MDSOC is to implement local unit tests in each hyperslice. The functionality tested is the local functionality of the hyperslice. When composition occurs, functionality can change. We have verified this when introducing the style check feature. Multidimensional unit testing, due to its' particularities, is definitely a field for future research.

We also implemented a simpler logging mechanism than the one proposed in [4], this time we used bracketing. Another feature proposed for this SEE in [3] is caching. Each expression would cache the result of evaluation for future usage. Cache invalidation would also be an issue for this concern. This could easily be attained if the `Eval()` method was bracketed with methods such that the cache contents were tested to check if these were usable, if so, instead of evaluating the expression, this result was returned. This could be done as a simple enhancement to Hyper/Net's bracket attribute to provide around functionality.

7. RELATED WORK

Hyper/J [5] has been available for download from IBM since 2000. We've already presented several Hyper/Net comparisons with it, noticing that Hyper/Net is not nearly as mature and is still much more limited. We've also verified this is not the case when it comes to decompose existing code as we are able to attain with Hyper/Net the same result as if the code had been initially developed with Hyper/Net support while with Hyper/J we cannot. At this point Hyper/Net still presents serious limitations regarding traceability for compilation errors and debugging; both will be shown relative to the generated code. Hyper/J is capable of manipulating pre-compiled code while Hyper/Net isn't, simply because Hyper/J operates at the byte-code level while Hyper/Net at the source-code level.

Interestingly, right after the first version of our prototype had been finished, [10] was published. [10] describes in detail HyperC#, a MDSOC prototype implementation for C#. [10] considers MDSOC as part of AOP, we don't consider such classification exact. HyperC# gathers meta-data through a GUI (graphical user interface) where a class is defined manually. HyperC# also works at the source-code level by means of a parser. After class declaration, the decomposition stage starts, this is also done in a GUI. The methods and constructors can then be moved into concerns, rendering these as the only units of composition. The same GUI provides functionality to define a default composition action (like Hyper/J) and insert specific composition rules using an equates construct (equivalent to Hyper/Net's merge) and a bracket construct (also present in Hyper/Net). HyperC# is limited to hyperspaces of one class file at a time, this limits its' current usage scenarios. Only the C# language is supported, there exists a previous work from common authors that implements MDSOC support in VB.Net.

AOP [7, 8] is closely related with MDSOC and, as already proposed, might provide an interesting source of experience for MDSOC works, especially in the field of composition rules. Aspects can also be used as artifacts and be an active unit in an MDSOC hyperspace, further investigation into such usage is required.

Step-wise refinement [9] is an approach with a lot in common with MDSOC; it works by incrementally adding features to existing simple programs.

8. CONCLUSIONS

We have started with an approach to MDSOC based on a simple and native artifact in both the C# and VB.Net 2.0 languages: partial types. We described our approach, presented a full example for it and discussed its' limitations. Some of these were addressed by our prototype

implementation, Hyper/Net. Hyper/Net's functionality was described. Three composition constructs materialized as programming attributes were discussed: overriding, merging and bracketing. Hyper/Net's architecture and implementation was also described. The initial example, left unfinished with the partial type's limited version, was extended with new features, while exposing some of Hyper/Net's limitations.
For both examples, we used unit testing to verify the correct result of our work and identified particularities with unit test artifacts in a multidimensional environment. A new paradigm, multidimensional unit testing, was proposed by defining a few simple rules that we applied with success.
All the areas covered here have serious future work still to be done. We compared our approach to the existing similar ones.

REFERENCES

[1] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns". ICSE, pp. 107-119, May 1999.

[2] H. Ossher, P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software". CACM 44, pp. 43-50, 2001.

[3] H. Ossher, P. Tarr, "Multi-dimensional separation of concerns in hyperspace". Technical Report RC 21452(96717)16APR99, IBM Thomas J. Watson Research Center, Yorktown Heights, NY., 1999.

[4] H. Ossher, P. Tarr, "Multi-Dimensional Separation of Concerns and The Hyperspace Approach". Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2000.

[5] P Tarr, H. Ossher, "Hyper/J User and Installation Manual". IBM Corporation, 2001. http://www.research.ibm.com/hyperspace.

[6] W. Harrison, H. Ossher, "Subject-oriented programming: a critique of pure objects". OOPSLA, pp. 411--428. ACM Press, 1993.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., Irwin, J. "Aspect-Oriented Programming". ECOOP, Springer-Verlag, 1997.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, "An overview of AspectJ". ECOOP, pp. 327-353. Springer-Verlag, 2001.

[9] D. Batory, J. Liu, J.N. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns". ACM SIGSOFT 2003.

[10] A. Hantelmann, C. Zhang, "Adding Aspect-Oreinted Programming Features to C#.NET by using Multidimensional Separation of Concerns (MDSOC) Approach", Journal of Object Technology, 5(4), pp. 59-89, 2006.

[11] A. Rashid, A. Moreira, J. Araújo, "Modularization and Composition of Aspectual Requirements". Intl.Conference on AOSD, Boston, Massachusetts, March 2003.