Hyper/Net: MDSOC support for .NET

Tiago Delgado Dias
Brisa Auto-estradas, S.A.
Ed. CCO, Quinta da Torre da Aguilha
2785-599 São Domingos de Rana, Portugal
tdias@ptsoft.net

ABSTRACT

Multidimensional separation of concerns (MDSOC) is an approach to analysis, design and code artifact modularization.

Instead of prioritizing a dimension of the modularization, as is done in Aspect Oriented Programming, with MDSOC all dimensions are equal. MDSOC also promotes traceability throughout artifacts (from analysis to code).

We present a basic MDSOC implementation model using .NET 2.0 Partial Types. We further extend such model to support the composition of methods, in a way similar to Hyper/J, introducing our own prototype, Hyper/Net. Additionally a multidimensional approach for unit testing is presented due to the particularities of unit test artifacts.

1998 ACM Computing Classification System: D.2.3, D.2.13, D.3.3.

Keywords

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

1. INTRODUCTION

Nowadays Object Oriented Programming is the most common paradigm for software development, especially in large projects, namely for product lines. Other paradigms like functional programming or logic programming are more common in specific niches. Also common in specific niches are Domain Specific Languages (DSLs), especially due to faster time-to-market for well defined types of problem and for the availability of RAD (rapid application development) tools for these DSLs.

Object oriented programming, or OOP for short, is centered on the class element. Classes usually abstract a physical entity or a concept and relate to each other just like the corresponding entities do. A similar approach, even though more limited, is taken when modeling databases where tables abstract the same concepts as classes do[1].

While classes are the central element in OOP, it's only due to the proven results of the modularization approaches used to define such classes that OOP made its' way to the current, easily observed, widespread. One developer can create a class, implement its' methods and offer it's functionality at a higher level, relative to the implementation. By composing such levels on top of each other it's possible to offer a very high level of abstraction where some of these methods can map directly to requirements that gave origin to the software development in the first place. Furthermore, through interface definitions (either the OOP native to the language or, for example, defined in a WSDL to be used in a Web Service), it is possible to map a definition of functionality to several different implementations, eventually interchangeable and potentially developed separately.

Even though, the same early adopters of OOP, now supporting their product lines on this technology, are facing several challenges and looking for solutions. While development from scratch with objects[2] can run smoothly and be easily maintained, as soon as we need to integrate such development with other systems not initially predicted or add new functionalities to it, issues start to arise.

We support that this happens because the software is initially developed to be optimally organized for its' purpose, eventually leaving room for a few predicted improvements. As practice dictates new features are usually not predicted nor expected and thus require being implemented throughout several classes that had once been thoroughly organized and separated from each other. This has two major negative effects: it scatters the code for the new features in the original implementation and tangles unrelated code in the original class implementations. As a result the addition of features becomes time-consuming and eventually risky[3], but, much worse, contributes to the lowering of quality of the original code, leading way for a cycle of code degradation and complexification as time (and enhancements) go by.

Refactoring provides some answers for problems emerging by the previously highlighted reasons, but this is while there is a possible object representation that is able to separate additional features from each other.

While refactoring was becoming more popular in the mid' nineties, another approach, Subject-Oriented Programming [6] was being proposed as a solution for exactly the problem stated. Subject-oriented programming is an approach based on OOP that proposes that classes are not implemented regarding a real-world entity, but instead capture only a partial view of that entity, relative to a subjective point of view (the subject, for example how a prey sees a fish). These partial views are then composed in order for subjects to cooperate; for example, when modeling the fish, even though knowing the fish is sleeping might not be the concern of a prey, it could be important to know the fish is

[1] And thus are prone to mapping from one to another.

[2] Although classes are the main element of OOP we usually refer to objects, its' instantiations. This might be due to the fact that there are usually several orders of magnitude more objects than classes in a running piece of OOP software.

[3] Even though, risk can be minimized by the utilization of adequate unit tests and the deployment of a wider test driven development platform.

sleeping when implementing the prey subject, as the fish is unable to eat preys at that point, thus a subject containing the sleeping behavior should be composed with the prey subject of the fish.

A few years latter Gregor Kickzales, et al. were proposing yet another solution (Aspect-Oriented Programming, or AOP [7]) for the problem at hand, this time rolling out a working implementation (for Java) of the concept introduced: Aspect/J [8].

Aspect-oriented programming acknowledges the ubiquity of cross-cutting concerns as the most significant barrier to obtaining better results with OOP. Cross-cutting concerns are just like features that need to be implemented in more than one object in an existing implementation, typical examples are logging and authentication.

AOP encapsulates cross-cutting concerns in modules called aspects which contain themselves the operations that would be scattered throughout the code. AOP refers to these operations as advice. To apply such operations to the sections of the original OOP code where they would need to be placed AOP introduces the concept of pointcut, which defines exactly that, where the operations (advice) should be applied. Implementations like [8] use this information to create basic OOP code in a process called weaving, where aspects are mingled with the original OOP code, giving origin to tangled code which will be only seen by the compiler.

Along the line of analogy of OOP with database structures we can see aspects as triggers on databases, defining additional functionality without mixing it directly in tables (objects). Naturally the database trigger model is usually very limited, occurring at a much lower level than aspects.

Another approach to solving the problem presented is Multidimensional Separation of Concerns (MDSOC). It's based on work from subject-oriented programming and was first proposed in [1] by the authors of subject-oriented programming among others. MDSOC instantiates some of the concepts presented by subject-oriented programming but overall defines a more complete platform with new concepts. Namely different subjects regarding the same concern (for example the same feature) are brought together in grouping elements, hyperslices, which are autonomous modules. Hyperslices are then composed, just like subjects, into hypermodules which can be combined and interact to create software products.

Hyperslices and hypermodules inhabit hyperspace, an N-dimensional space where each dimension is divided in discrete and disjoint sections: concerns, instantiated as hyperslices. Common dimensions are objects, features, security, aspects; each dimension can be supported on its' own artifacts (objects, aspects, functions, etc.). Having more than one dimension enables capturing new software features, and more generally all evolutionary elements [2], in the artifacts chosen and each separated in its' own hyperslice/concern for the dimension, thus not dependent upon each other.

It's also possible to use only one kind of artifact and even then have multiple dimensions; such provides a solution for software evolution with OOP that doesn't require using additional artifacts or abstractions. This is the scenario supported by Hyper/J, an MDSOC implementation for Java [5].

Our focus on the remainder of this paper will be on MDSOC, starting with an overview in Section 2. Section 3 presents .NET's partial types which support our initial approach to MDSOC on this platform. Section 4 presents a classic example of MDSOC based software development using only partial types. Section 5 discusses the limitations of partial types for MDSOC support. Section 6 introduces Hyper/Net, our tool, which extends partial types for further MDSOC support. Section 7 exemplifies further implementations to the example from Section 4, impossible with the basic support from partial types. Section 8 focuses on the current Hyper/Net limitations and future work branching from our approach. Section 9 quickly spans related work, leading way to the conclusion in Section 10.

2. MULTIDIMENSIONAL SEPARATION OF CONCERNS (MDSOC)

In the early stages of software development several constructs exist to capture the ambit and properties necessary: requirements, use cases, functional specifications, etc. These constructs have in common the fact of capturing a piece of the problem, thus are serving to decompose the problem into smaller ones. Pieces regarding similar problems are usually grouped as in UML packages, viewpoints, etc. In MDSOC, concerns are generic grouping elements and can be mapped to any of these or other existing grouping elements. Thus the granularity of concerns varies but in most MDSOC approaches concerns tend to be generic.

Each concern in MDSOC exists in the context of a dimension. Dimensions are more general aggregators. Common approaches, like OOP, evolve in only one given dimension, namely the object dimension. Aspect-oriented programming or other approaches can add another dimension to the concern space, but these work as a complement to the object dimension which is still at the core, namely in AOP, the aspectual dimension is complementary. This limitation is referred to in [1] [2] [3] [4] [5] as "tyranny of the dominant decomposition". In order to break such tyranny, in MDSOC 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, in other words a specific dimension doesn't have any cross-cutting concerns regarding itself. Overlapping concerns may exist in between dimensions, but such cross-cutting is already sliced at the dimension level. Concerns are composed of artifacts (instantiations) or units, for example objects, these can span several concerns in different dimensions but never in the same dimension.

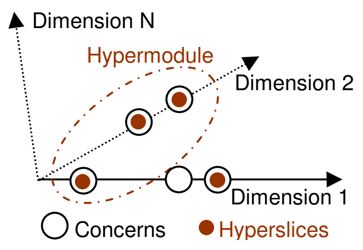

Figure 1. An hyperspace representation.

The space defined by the dimensions identified is referred to as the hyperspace 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 cross-cutting it is necessary to decompose each of these units and only include in the hyperslice the component that is related with the concern in question. Unit decomposition is done during hyperspace definition, this can either be done on existing software or be part of the development process itself.

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 cross-cut several dimensions, as can be seen in Figure 1, and can also be composed with and on top of on each other to attain complete software systems.

By organizing our hyperspace in concerns and dimensions the MDSOC approach can be applied from the first software analysis stages to the actual implementation, dealing with the native artifacts at each level of Software Engineering. Furthermore, artifacts at each level are organized in the same concerns and dimensions, thus mapping from one level to the other becomes trivial and direct, thus promoting traceability. There are approaches with similar aims for AOP, namely [11], but these require the introduction of new artifacts to the analysis stages (namely the concept of aspect) while MDSOC relies solely on existing native artifacts.

Nevertheless, our work with MDSOC is limited to the programming/code level, so, next, we present our initial approach based on partial types, a .NET language construct.

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. Both languages were created for use with the .NET framework which was deeply based on the Java environment. Currently .NET has gained its' position just like the Java environment, each with its' own worldwide community of adopting programmers and each undergoing thriving evolution.

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.

A common usage for this feature is separating tool-generated code from human-generated code for the same class, enabling easy regeneration of the tool-generated code without ruining the human-generated portion of the class.

Here we explore the usability of partial types to organize code by adopting the multidimensional separation of concerns (MDSOC) approach.

Throughout this paper we will consider hyperspace units at the class/object level. As seen in the previous section the first step in applying MDSOC is defining the hyperspace. First off dimensions and concerns are determined; notice that these can easily be extended afterwards as new additions are trivial. To populate the dimensions existing units must be decomposed or new units must be created, with declarations scattered in the concerns that are focused by the unit.

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 available to any programmer; 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, simply because partial types require each partial class declaration to be in the same namespace.

Notice that in our model the source code is manipulated while decomposed, offering the programmer full MDSOC support as soon as decomposition is done. When using Hyper/J to apply MDSOC to existing Java code it won't output the modularized code but requires managing the abstract code model defined in the hyperspace declaration file against the original scattered and tangled code. When working originally with MDSOC in Hyper/J the source should be organized by concern and the MDSOC model is then manipulated directly by the developer. The same happens with our approach, but for us, applying this model for developing from scratch or to an existing piece of software results in exactly the same multi-dimensionally organized code.

The second stage in implementing MDSOC, 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 latter on.

4. THE EXPRESSION SEE

As an experiment of the proposed model we implemented a classical example that accompanies MDSOC in [1] [3] [4] and comes as demo with Hyper/J [5].

The expression software engineering environment (SEE) is a simple OOP implementation of mathematical expressions with numbers, operators (+, - and assignment) and variables. It supports such features as printing expressions, evaluating an expression value and checking the syntax of an expression.

This hyperspace will have only two dimensions, the object dimension and the feature dimension which we will be working at. The object dimension is created by default in Hyper/J, usually we only work at the level of the other dimensions, Hyper/J also introduces concepts as "None" concerns in each dimension where units regarding no existing concern in the dimension are placed, we have discarded these in our approach.

First of we created a directory for a Kernel concern, the kernel concern is a basis concern where we declare each class and offer basic functionality: constructors, related private variables and eventually get/set methods (but we have none). We use the Kernel concern to organize our class hierarchy and do so in the following way:

- An Expression is an abstract super-class for all the other classes.
- Binary operators share commonalities captured in a class of their own (BinaryOperator) which derives from Expression. Sub-classes 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.

Another concern (Display) focuses on printing expressions on the screen, so, in another directory of the Feature dimension (root directory) we have the previous classes with only the implementation of a `Display()` method. This method simply prints on the screen a representation for the object, for example the Number object prints the integer value with which it was initialized. We verify here that elements from the Kernel concern are required for this concern; the typical approach followed in Hyper/J would propose the declaration of stub methods/variables for each of the elements required from other hyperslices, that latter would be composed (in an hypermodule) with the real elements. Here we use the variables from the other concern directly; this approach is discussed in the next section where a better solution is proposed.

The display hyperslice also contains the definition of unit tests regarding the display method. Testing here is not a concern by the definition presented, if it was it should be factorized into its' own directory, isolated from each of the other concerns implemented. By placing separate unit tests in each hyperslice we are optimizing the system for mix-and-match operations[4], 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[5]) and compiling a new version of the project. Notice that unit tests for the feature remove are also removed, if Testing was implemented as a concern we would have to remove manually tests for the removed concern in order to be able to compile the project.

The Evaluation and Check concerns are implemented much in the same way as the Display concern, each 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 three child classes.

[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). We will analyze this and other limitations in the following section.

The sample code for this example is available for download from http://ptsoft.net/tdd/.

5. PARTIAL TYPE LIMITATIONS

In our partial type approach, elements of a class declared in a specific concern are available throughout the remaining declarations of the same class for all other concerns[6]. This will require special care from the programmer, once an element from another concern is referenced a tight bound has been introduced and changes to the element become cross-cutting (having to change the way it's referenced).

This happens because hyperslices with partial types are not declaratively complete as proposed in [3] and supported in Hyper/J. Declarative completeness means that a hyperslice is not dependent on other hyperslices, with Hyper/J this usually means that a hyperslice contains abstract class declarations which hold abstract declarations for elements not internally provided. The hypermodule then requires a composition rule to override the abstract element declarations with real-ones provided by other hyperslices.

In both approaches when an element in a concern that is used by other concerns changes, these changes must be propagated. In the case of Hyper/J and its' approach changes must at least be made to the composition rules, if the change is serious enough the usage in each other hyperslice must also be reviewed. With partial types the usage from other hyperslices will always have to be reviewed and there is no centralized information regarding such usage. With the extensions provided by Hyper/Net, presented in the next section, we can easily implement a similar approach to that of Hyper/J.

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

[5] 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/.

[6] When using a .NET 2.0 compliant IDE, the auto-completion feature will suggest all class elements, even those that were defined in a different partial class declaration.

We consider that neither approach is sufficiently satisfactory and further investigation in this matter is required. The relations between hyperslices could be stabilized in an interface, eventually a public interface provided by each hyperslice, but the effects of such are unpredicted and stabilization risks making changes to hyperslices harder.

The last change examined in our example was impossible to implement given the natural limitations of partial types. 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.

6. 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 approach.

Hyper/Net builds on this initial work by providing explicit composition constructs that elevate methods[7] 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 by having one replace a set of others and merge by capturing the functionality of each instance in a new 'super-instance'. Bracket enables inserting actions occurring before and after a specific method, this construct is similar to AspectJ's [8] `after` and `before` advice and was already present in Hyper/J.

Overrides is the simplest composition construct, only one of the conflicting methods can define this attribute and that 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. 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 rule in Hyper/J with the priority argument (which is optional[8]). 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. Such method must be local in the class and receives a list of result objects to return only one object of the same type[9].

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

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

The bracket constructor 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[10]. 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

[7] 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.).

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

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

[10] 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.

Hyper/Net uses the parser implemented by NRefactory[11] to produce an AST (abstract syntax tree) for the code.

Following the initial processing stages there is a composition preparation stage. It's in this stage that partial types are merged into a single type declaration, but before there is a similar merge that is done with namespaces; bring 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[12] 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; at this point matching names are considered to be the same method, real matching should be done by comparing method signatures also. Remember that scattered methods have already been brought under the same tree elements in the AST by 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. Following 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.

7. 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 we were unable to introduce with only partial types, we now can 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 the name of variable elements so these are smaller than 5 characters. This way our concern would only require a default implementation for expression, returning true, an implementation for the binary operator making sure each side expression is correct and finally the implementation for the variable. Additionally unit tests should

Each of these methods has a `MethodMerge` attribute declaration applied to it. The action defines the merge as the method merge action, a priority level inferior to the default merge priority which is the one of the syntactic check and a method that 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 4. we can recall that in this dimension 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. While adding the style check feature we had to add another initialization method for the test class, now the tool[13] we are using for unit testing supports only one initialization method for each test class. We didn't have to change our code to support this limitation; we simply added a merge attribute to the initialization method on the second test class it was defined, this was transparent for both the programmer and for the unit testing tool.

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

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

[13] We used NUnit (www.nunit.org) for unit tests. NUnit is already integrated in the SharpDevelop IDE.

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. The style check test allows for assignment of two binary operator expressions, which is not allowed by the syntactic check. The style check concern tests expect a valid result when combining such two binary operators by assignment; the syntactic check tests expect an invalid result for the same expression. When run separately (by removing the other check feature by mix-and-match) all unit tests pass, when run with the two features coexisting the style check test for the binary operator fails as the functionality being tests is now the combined functionality of both checks. First of all unit tests should be able to run on independent hyperslices and support for that should be provided by the IDE environment itself, manually we can remove unwanted hyperslices so we can test other in isolation. Then, in order to test combined functionality, tests should be composed. Composition for this purpose most likely should target lower level units inside methods. 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. A new hyperslice was introduced (Logging), this hyperslice only contains a partial class definition for Expression. Two new methods are introduced: `LogMethodEntry()` and `LogMethodReturn()`. These methods become available to all other classes through inheritance and so we can now add logging to any method by adding the attribute in Figure 10. Each method is kept as a local delegate instance for valid type-checking when processing the attribute.

```
[MethodBracket(LogMethodEntry, LogMethodReturn)]
```

Figure 9. Attribute declaration for the logging feature.

Declarative completeness can now be attained by declaring stub methods local to each hyperslice, and then overridden by respective methods in a different hyperslice. This way we can remove cross-cutting references from our hyperspace, even though, as stated previously we would like to find a better approach.

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 we usable, if so instead of evaluating the expression this result as returned. This could be done as a simple enhancement to Hyper/Net's bracket attribute to provide around functionality.

The sample code for this example is also available for download from http://ptsoft.net/tdd/, along with a prototype version of Hyper/Net.

8. FUTURE WORK

Several concerns for future work have already been identified throughout this paper. We now focus some of these and present the more ambitious areas for future research.

Hyper/Net still has a long way to go in terms of composition expressibility. Extensions to the present constructs are required; a few have already been pointed out in this paper, an important one that remained unmentioned is the support for pattern matching or similar functionality for the bracket construct. Units for composition need to be generalized from methods to other class elements and then drill down inside method implementations. AspectJ's approach can provide important experience in this field; we have already identified several semantic similarities.

Multidimensional unit testing has already been identified as a very interesting field for future research, eventually to be extended to the entire test driven development universe.

In Hyper/Net composition rules are distributed/scattered. This is not necessarily bad; composition information can be precious when present at each hyperslice. But naturally there should be a centralized way of manipulating the compositions, for such we propose, instead of a centralized meta-data base (like Hyper/J's concern mapping files), an hyperspace management tool that operates by merging the distributed definitions. Such a tool could be part of the development of IDE support tools for Hyper/Net.

Another interesting IDE enhancement would be a tool that enables programming directly by viewing a specific dimension of the hyperspace, for example, when adding new objects we should be viewing from the perspective of the Object dimension, not the Feature dimension. We could then use our Hyperspace management tool to define meta-data for the newly added artifacts. If we were coming from an upper level process (with a design in hand) we could have that automatically done for us, just like class signatures are. The programmer should be able to view his code from different perspectives, further being able to program using those perspectives.

A limitation of Hyper/Net that is common to Hyper/J is the necessity of recompiling code in order to add new concerns (or even dimensions). The possibility of redefining the concern space without recompilation was proposed in [6]. In previous (unpublished) work we have used an event and hook based system that enables plugging in new components that use these events or hook without the necessity of recompilation. Such approaches might serve as a basis for adopting plug-and-play edition functionality to Hyper/Net.

Also based on the work from [6] we find it uninteresting to limit the output of our MDSOC approach, in the case of developing software that offers programmable interfaces (APIs), to generating an agglomerated version of the composed concern-space. Instead we propose an additional step, after composition, which will create a set of interfaces for each class, each comprising the public functionality of the class for a given concern. This might be an instantiation of the subjective views of [6]. The usage of such interfaces should be further studied, namely its' mapping to documentation, keeping the MDSOC approach available all the way through the development process (documentation included).

As can be seen, the example adapted from earlier work on MDSOC (and the most important to date) contains only one dimension. Other dimensions could easily emerge if logging had to be more specialized or by adding new cross-cutting requirements. Managing the multidimensional space becomes more critical as more dimensions are added, and here we (humans) may have a particular limitation inherent to the basis of the approach. Human comprehensibility of a number of dimensions greater than 3 is usually limited. In MDSOC we're not usually required to consider the full dimension set at once, but there may be situations which require agile multidimensional thinking, namely when defining ways to organize an hyperspace, this might explain why most examples of MDSOC are not rich in terms of dimensions.

9. 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. 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 instead. 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, we had access to [10] which describes in detail HyperC#, an MDSOC prototype implementation for C#. [10] considers MDSOC as part of AOP, we don't agree with such classification. HyperC# gathers meta-data through a GUI (graphical interface) where a class is defined manually. This GUI introduces interesting visualization of a class and might provide very useful if integrated in an IDE. HyperC# also works at the source-code level by means of a parser, just like Hyper/Net. After class declaration, the decomposition stage starts, this is also done in a GUI which has two sections, one where dimensions and concerns are managed and another listing the classes methods. 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 (like Hyper/Net's). While these constructs already provide method signature support unlike Hyper/Net no additional feature support (like ordering or result merging) is provided. HyperC# is limited to hyperspaces of one class file at a time, this limits very much its' current usage scenarios. Only the C# language is supported, there exists a previous work from common authors that implements 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.

10. SUMMARY

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, of extensions for full MDSOC support in these languages. Hyper/Net's functionality was described; three composition constructs materialized as programming attributes: overriding, merging and bracketing. Hyper/Net's architecture and implementation was described. The initial example, left unfinished with the partial type's limited version, was extended with new features, meanwhile 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 focused some of it. We also compared our approach to the few existing similar ones.

11. REFERENCES

[1] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton, *"N Degrees of Separation: Multi-Dimensional Separation of Concerns"*. Proceedings of the International Conference on Software Engineering, pp. 107-119, May 1999.

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

[3] Harold Ossher and Peri 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 and P. Tarr, *"Multi-Dimensional Separation of Concerns and The Hyperspace Approach"*. In Proc. of the 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] William Harrison and Harold Ossher, *"Subject-oriented programming: a critique of pure objects"*. In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 411--428. ACM Press, 1993.

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

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *"An overview of AspectJ"*. In Proceedings of the 15th European Conference on ObjectOriented Programming, 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] Angela Hantelmann, Cui Zhang, "*Adding Aspect-Oreinted Programming Features to C#.NET by using Multidimensional Seperation of Concerns (MDSOC) Approach*", in Journal of Object Technology, vol. 5 no. 4, pp. 59-89, May-June 2006. http://www.jot.fm/issues/issue_2006_05/article1

[11] Awais Rashid, Ana Moreira, and João Arajo, *"Modularization and Composition of Aspectual Requirements"*. 2nd International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, March 2003.