UNIVERSIDADE NOVA DE LISBOA

Faculdade de Ciências e Tecnologia

Departamento de Informática

Hyper/Net: MDSoC support for .NET

Por

Tiago Delgado Dias

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática.

Orientador: Professora Ana Moreira

Lisboa

(2007)

Agradecimentos

Gostaria de agradecer a quatro pessoas sem as quais não existiria esta dissertação:

- Ao Jorge Lopes, por acreditar em mim, por me ter motivado para o mestrado e por ter criado as condições laborais que mo permitiram realizar durante dois anos que também foram de trabalho intenso na Brisa.
- Aos meus pais, Gabriela Delgado e José Dias, pelo apoio constante e incondicional.
- À Professora Ana Moreira, pelos aconselhamentos essenciais e pelo empenho dedicado na revisão de todo o meu trabalho.

E pelo interesse e apoio demonstrado, e por nutrir um ambiente dinâmico de descoberta e iniciativa no seio da Brisa, agradeço especialmente ao Professor João Bento.

Sumário

Esta dissertação usa a Separação Multi-Dimensional de Assuntos (MDSoC) para estudar a composição de software. A composição de software endereça as dificuldades de modularização típicas que existem nas abordagens actuais da Engenharia de Software, como por exemplo as Orientadas aos Objectos. O MDSoC oferece o mesmo mecanismo de modularização multi-dimensional unificado para todas as fases do ciclo de vida do software. Este mecanismo de modularização complementa as abordagens existentes, em vez de as substituir.

Grande parte do trabalho aqui apresentado foca implementações MDSoC limitadas à fase da programação. A este tipo de implementações já existentes junta-se o Hyper/Net que desenvolvemos de forma a suportar MDSoC no ambiente Microsoft .NET. O Hyper/Net baseia-se nos tipos parciais (do inglês *partial types*), que são uma funcionalidade nativa das linguagens .NET. A utilização de funcionalidades nativas das linguagens para o MDSoC é uma inovação e é possivelmente uma das contribuições mais interessantes deste trabalho. Como forma de validação, o Hyper/Net foi analisado à luz do modelo MDSoC e comparado com outras implementações. Foi também utilizado na implementação de casos de estudo simples, que mostram os benefícios do MDSoC. Finalmente, os resultados de cada caso de estudo foram validados através da utilização de testes unitários, adaptados ao MDSoC.

Abstract

This dissertation uses Multi-Dimensional Separation of Concerns (MDSoC) to focus on software composition. Software composition emerged as a response to difficulties found in modularization with standard Software Engineering approaches such as Object Oriented approaches. MDSoC provides a unified multi-dimensional modularization mechanism that is usable across all the stages of the software lifecycle. This modularization mechanism complements the existing approaches, instead of replacing them.

Most of the work presented in this document addresses MDSoC implementations for programming. We developed such an MDSoC implementation for Microsoft .NET and called it Hyper/Net. Hyper/Net is based on partial types, which is a native feature of .NET languages. Relying on native language features for MDSoC is a novelty and is possibly the most interesting contribution of this work. To validate Hyper/Net, it was analyzed in the light of the MDSoC model and compared with other MDSoC implementations. Hyper/Net was also used to implement simple case studies that show the benefits of MDSoC. Finally, the results of each case study were validated by a unit testing approach, which was adapted for MDSoC.

Life hanging from petals

A seed into the ground
taken there by a myriad small things
growing into time
roots spreading and everything.

First a flower,
small, shy, fragile,
managing energy against entropy,
to grow wider, deeper in color,
scenting all around the tree,
and also everyday easier to see.

Petals have their time to be,
a fruit was an aim
and the flower just something to see?
Many surely disagree,
but the juicy fruit,
the aim,
is what ends up showing itself to me.

Picking it could come from necessity
but if weeks before
reaching at the same tree
you would be picking
out of curiosity
and, well, then the fruit wouldn't be.

Knowing not of words,
not even capable of planning,
something simple,
like planting another tree.
Both fruit and flower serve their purpose
and I figure Shakespeare could say "they be".

TheDruid (2003)

Symbology and Notations

- *Italics* are used to introduce new concepts and also to represent entities from the programming artefact of our examples (methods, classes, etc.).
- Closed braces [] are used to identify bibliographic references, which are listed from page 134 onwards.
- The `Courrier New` font is used to present snippets or entire blocks of source code.
- This dissertation is organized using a chapter – section – subsection hierarchy.

Contents

Figures

Listings

Tables

Chapter 1

Introduction

Software Engineering, as a branch of Engineering, provides a body of approaches that can be used to develop, operate and maintain software in a systematic, disciplined and quantifiable fashion [IEEE90]. These approaches break-up the process of software development into different, interrelated stages. Each stage in a software development approach is usually addressed by a different field of Software Engineering. This allows each field to focus particular aspects of the development process and allows the clear separation of the different needs that arise during software development. This separation is adequate for development because most approaches complete one stage before moving to the next, allowing the gradual detailing of the software pieces as they are developed.

This would be perfect if development was a completely planned task, done in a single iteration. On the contrary, most software has to evolve after its initial development is considered finished. Software evolution affects most stages of the Software Engineering approach that was applied. Evolution will require revisiting each stage and working in the context of the existing software elements. In fact, most problems in Software Engineering arise during evolution, after systems are initially developed and deployed. 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 or expected and, thus, require being implemented into the existing structure that was not designed for them. As a result, adding features becomes time-consuming and, eventually, risky. Even worse, it directly contributes to lowering the quality of the original implementation, leading way to a cycle in which software becomes degraded and more complex, as time, and enhancements, go by [Lozano06]. This cycle also causes an increasing difficulty in making changes to the software.

It is true that risk can be minimized by using adequate tests and adopting a test driven development approach. Still, testing does not help avoid the degradation of the implementation. It is also true that many changes that are not specifically planned can be supported using general extension approaches, like those provided by some design patterns. But, supporting changes comes at the cost of implementing these extensions during initial development. Furthermore, not all changes will be supported by these extensions.

Refactoring can be used to mitigate the degradation of existing software implementations, when new features are introduced, by adapting the existing software organization to the new requirements. Still, it would be desirable to support the new features required by evolution as

if they had been introduced during the initial development. None of the previous solutions can fully achieve this requirement. Yet, this is achievable if software captures the features alone and then integrates them with each other. This way, when introducing new features, the existing features do not need to be changed, but, only their integration does. This is the principle behind a set of solutions called composition solutions. This document focuses on one such solution in particular: Multi-Dimensional Separation of Concerns (MDSoC). As we will see, MDSoC is especially important because it supports the same multi-dimensional organization of software across the different stages of development.

1.1 Goals of this dissertation

The main goal of our work was to be able to use MDSoC while programming in .NET languages, in particular C#. It should be possible to use existing development infrastructures and as little as possible should need to be changed in the development process, apart from adopting an MDSoC approach.

After achieving this first goal, the adequacy of the MDSoC features possible with .NET languages should be validated. To do this, a classic example in MDSoC literature, the Expression SEE, should be implemented using .NET languages and MDSoC.

Finally, to guarantee that the example worked properly, it should be adequately tested for local functionality as well as overall functionality and the different possible combinations of functionality achieved by removing particular features.

1.2 Contribution of this work

As stated, our main goal was to implement MDSoC for Microsoft .NET, which is a multi-language programming environment. Other contributions were gradually achieved in a natural way as we used, evaluated and contextualized our MDSoC implementation. Namely, by providing examples of MDSoC usage, introducing a testing approach tailored for MDSoC and comparing existing MDSoC implementations.

To implement MDSoC for .NET, first, we identified important separation of concerns capabilities in partial types, which is a native feature of .NET 2.0 languages. This enabled us to develop a simple MDSoC approach using partial types (Section 6.1). This approach requires no other software aside from the Microsoft .NET framework and its standard development tools. This may be considered an innovation as most composition solutions require additional software, other than the original language compilers. The partial types approach is limited to composing types, which limits its abilities to separate concerns. It was extended to support method composition by developing Hyper/Net (Section 6.2).

With Hyper/Net we were able to develop small examples where the benefits of using MDSoC are clear. The examples themselves, in particular the ones presented in Sections 7.4 and 8.2[1], provide a small contribution, as they differ from the traditional MDSoC examples one can find in the existing literature. Due to time constraints and the prototype nature of Hyper/Net, it

[1] Hyper/Net was implemented using the partial types MDSoC approach and its implementation is presented as an example of MDSoC usage.

was not possible to use it to develop a real-world application, which could provide further support regarding the benefits of MDSoC.

The need to test our small MDSoC examples motivated us to explore a testing approach that is adequate for MDSoC. This testing approach is presented in Section 7.3, as part of a chapter that addresses the more pragmatic aspects of the partial types approach and Hyper/Net.

Finally, existing MDSoC implementations are presented in Chapter 5 and compared with Hyper/Net in Section 9.2.

1.3 Document structure

This dissertation can be divided into three parts. The first part, from Chapter 2 to Chapter 5, provides background information on composition solutions and existing MDSoC implementations. The second part, from Chapter 6 to Chapter 8, presents our own MDSoC implementation: Hyper/Net. The third part, Chapter 9 and Chapter 10, wraps-up the dissertation by comparing all the MDSoC implementations, providing conclusions and presenting future work. Finally, the document closes with Apendix I, where the Hyper/Net source code is listed.

It may also be adequate to separate the first two chapters (Chapter 2 and Chapter 3) from the following chapters, as they present the models of composition solutions. The following chapters (Chapter 4 to Chapter 8) address MDSoC composition implementations rather than the model.

This document starts by presenting two composition approaches. The first, Subject Oriented Programming (SOP), is presented in Chapter 2. SOP proposes a multi-perspective view of OOP, with different classes implementing different views of the same object. Next, Chapter 3 presents the MDSoC model as an elaboration of the SOP model. MDSoC introduces a structure for holding the views of the SOP model and provides a unified multi-dimensional view of the different stages of software development.

The remaining of this document focuses on MDSoC implementations. Chapter 4 provides the necessary technical background regarding the languages and platforms used in the MDSoC implementations. Chapter 5 presents two different MDSoC implementations, one targeting the Java language (Hyper/J) and, another, the C# language (HyperC#). Chapter 6 presents two MDSoC approaches that were developed by us: the .NET partial types approach and Hyper/Net. They are evaluated from the perspective of the MDSoC model presented in Chapter 3. Chapter 7 is targeted at developers and architects and describes how both of our MDSoC approaches can be used. It also provides details on how to test MDSoC programs. It ends with two complete examples of simple programs developed using our MDSoC approaches. Chapter 8 describes the architecture and implementation of Hyper/Net from an MDSoC perspective.

Chapter 9 compares the different MDSoC implementations according to architectural, structural, compositional, usage and reuse criteria. Finally, Chapter 10 provides conclusions on our work and identifies possible directions for future work, both for Hyper/Net and for the remaining focus points of this document.

Chapter 2

Subject Oriented Programming

Subject-Oriented Programming [Harrison93] was introduced in 1993. Subject-Oriented Programming (SOP) extends the Object Oriented Programming (OOP) model by sub-dividing class-hierarchies into subjects that latter are composed together. Subjects can be seen as containers for classes under a specific point-of-view or perception of the world. A class, which usually agglomerates several different points-of-view in OOP, will give away to several different classes in SOP, each located in the appropriate subject container. Subjects act solely as containers and the classes contained by them will still be constructed in a fully OOP manner. While some subjects will be useful in a standalone fashion, most require composition with other subjects to extend their usefulness.

This chapter starts by presenting the SOP model according to [Harrison93]. Next the SOP composition model is analysed in more detail, based on the SOP composition model as presented in [Ossher96]. After that, the benefits of SOP for software engineering are overviewed. Finally, Subject Oriented Design, an extension of the SOP approach for UML design diagrams, is presented.

2.1 The SOP model

SOP allows the separation of classes into different subjects. In order to be useful, most of the classes inside these subjects must be composed. One of SOP's composition mechanisms is additive composition. Additive composition combines all aspects of the composed elements. SOP supports a particular additive composition, *merge* composition [Harrison93]. With merge composition, if different subjects implement the same method for a composed class, when the method is invoked, all corresponding methods in each composed subject will be invoked in turn. A different composition method, *nesting*, is also proposed in [Harrison93]. Nesting uses the order of compositions to create scopes in which to invoke methods. The more recent compositions are the ones that are readily available for method invocation.

Composition enables instances of classes in different subjects to be correlated. At runtime this is done through identity elements called *oids* (object identifiers). If two classes from different subjects are composed together, by creating an instance of one of these classes, an instance of the other will also be created. These classes are related by a common *oid* that is allocated during initialization. Two initialization models are proposed in [Harrison93]: an immediate

and a deferred initialization model. Deferred initialization offers better performance than the OOP equivalent of the composed subjects, especially when several subjects are being composed. In immediate initialization all classes associated with the newly created *oid* are initialized at the same time during object initialization. In deferred initialization, classes associated with the newly created *oid* are only initialized the first time a method or variable they offer is accessed. Deferred initialization has a requirement that constructors do not have any parameters. According to [Harrison93] the absence of construction parameters is also beneficial for adding new subjects without the need to change class construction, independently of the initialization model. Otherwise, if constructor parameters are used, constructor composition would become more complex (or even impossible) in order to support deferred initialization or adding new subjects dynamically.

Composition requires that classes are matched between different subjects. One simple method is explicit matching by class names. Even this simple matching mechanism enables subjects to have exclusive classes that match no other subjects. This allows each subject to hold its own class hierarchy, having only a subset of its classes that explicitly match other subjects' classes. Sometimes there is also advantage in matching some of these classes that have no explicit matching. [Harrison93] explores some non-explicit matching methods.

When different class hierarchies are being combined, there are several detriments to keeping each existing inheritance structure while matching different class hierarchies. First of, there might be no semantic value attributable to the combined inheritance structure; the hierarchies might make sense only in the context of each subject. Furthermore, the resulting class hierarchy might exhibit cycles and the diamond problem. Finally this approach limits the composition of subjects from different languages as semantic details with their own inheritance mechanisms would have to be integrated (for example multiple-inheritance). As such, [Ossher96] introduces the process of flattening class hierarchies prior to composition. Flattening is the process of copying all inherited methods and variables to the classes being matched and removing all inheritance information. This way, there is no class hierarchy in the composed result. This might be an acceptable result, especially taking into account [Harrison93] advises against extending the composition result itself. [Harrison93] suggests that all work should be done in each subject prior to composition.

While creating or extending an application with SOP, subjects may be added as needed. For instance, if a certain class starts to grow in a particular subject and parts of the class do not fit the subject's scope anymore, those parts may be factored out into an appropriate new subject. As such, subjects can start-out with only a single class and then grow as other classes fit the subject. This way some subjects might be more important than others: one such subject is the "intrinsic" subject referred by [Harrison93]. An "intrinsic" subject usually captures internal state and essential behaviour of a particular object, with other subjects capturing elements from the perspective of external viewers of the object. In most cases, the "intrinsic" subject will become the dominating subject in a SOP application. This might be limiting to other subjects but in fact is not, as SOP enables each subject to have its own class hierarchy, independently from the "intrinsic" subject hierarchy. Nevertheless [Harrison93] leaves it up to the designer or programmer to determine if an "intrinsic" subject should be used or if a more equalitarian subject space is appropriate.

2.2 Detailing the SOP composition model

A more formal model for SOP is also presented in [Harrison93]. In this model composition is presented as a tuple *(R, Q)* where *R* is a composition rule and *Q* is a set of components. These components can be either subjects or other compositions, making composition rules in this model recursive. [Harrison93] does not detail the nature of composition rules *R*, but, to allow reuse, these should be general (like merge), unbound to subject elements. In such case, there is a major limitation in this model as it only enables the composition of entire subjects and does not offer constructs to compose elements from each subject in different ways. The granularity of the composition mechanism defines its power. Versatile composition mechanisms that are able to operate simultaneously at different levels are the most powerful solution. [Ossher96] introduces such a model for subject composition. This model overcomes the limitations of the [Harrison93] model by using general rules together with a set of exceptions clauses. Exception clauses enable defining special composition behaviour at a finer granularity than the rules which operate at the entire subject level. In fact, general rules are created in a modular fashion from the same clauses which are used for expressing exceptions. Many aspects of the general rules construction are parameterizable when general rules are used. As such, the model introduced in [Ossher96] is customizable, by allowing the parameterization of existing rules, and extensible, by allowing the creation of completely new rules.

[Harrison93] does not define a structuring mechanism for the subject space. Nevertheless, the hierarchic nature of composition (due to its recursiveness) offers some structuring support. Different orders in the hierarchy of composition can be used to structure our subjects differently. But relying on composition to structure a subject space is too limited; the subject structure will always be influenced by the semantics of composition. This may not be an issue in small subject spaces, but for larger subject spaces, SOP lacks a specific subject structuring mechanism.

2.3 Software Engineering benefits of SOP

[Harrison93] also presents implications of SOP in software engineering. New subjects can be added to running applications without requiring their recompilation. This is an important technical achievement with several implications in project development methodologies. By supporting runtime extension of software, features can be deployed incrementally as they are designed, developed, composed and tested, without affecting the availability of the running environment. SOP also provides increased encapsulation as composition is external to subjects, so these are independent and need not be changed for integration. This enables the standalone creation and evolvability of subjects without imposing it. It is always possible to develop tightly coupled subjects or loosely coupled subjects with particular compositions in mind.

[Ossher96] introduces the idea of composing subjects written in different languages. This is also a very important concept because it would enable the choice of the most appropriate language to implement a particular subject. This way, some subjects could use logic programming while others OOP or other appropriate paradigms.

[Ossher96] also proposes the usage of subjects for separating the code according to the different requirements identified during analysis. This is an important step as it will enable a direct mapping of requirements to design and code as presented in [Clarke99].

2.4 Subject Oriented Design

SOP is tailored for the modularization of code. Nevertheless, subject containers seem adequate to be applied to other artefacts of the software lifecycle, namely design and analysis artefacts. This is especially true if taking into consideration the proposal of using subjects to directly map different requirements [Ossher96]. As such, [Clarke99] introduces a Subject Oriented Design (SOD) model.

To understand the need for SOD, [Clarke99] introduces several issues in mapping requirements to object oriented design. The artefacts involved in the software lifecycle have their own decomposition mechanisms. Though, most artefacts are limited to one kind of decomposition. Typical analysis processes are based on functional decomposition by requirements. Object oriented design and programming both lack a functional decomposition mechanism but instead provide object decomposition. Object decomposition is useful as it can match concepts of the application domain and categorize them. But artefacts are not isolated, there is need to map analysis requirements to design and then to code. If such mapping is effective, requirements can be traced throughout artefacts. In fact, other kinds of decomposition, namely objects, can also be used for such tracing (usually from code to design and requirements). In reality, because artefacts have different and incompatible decompositions, a direct mapping is impossible. Functional components (namely analysis requirements) are scattered throughout several objects (in object oriented design and code) and different objects are tangled within requirements. Direct traceability between artefacts is thus impossible when the artefact decomposition mechanisms do not match.

Before introducing the SOD solution, [Clarke99] tries to use the object decomposition, in particular inheritance, to create functional decompositions for a specific example. In [Clarke99]'s example, there is a requirement that maps to extending a particular method, shared by several classes in a hierarchy. To implement this requirement in a decomposed fashion, each existing class is extended using inheritance. Each extended class will have an override of the existing method that implements the new requirement, also invoking the existing functionality. After the addition of a few features, using this kind of decomposition, the class hierarchy will have grown several times in size. At this point, the addition of new class elements will also require the creation of all the subclass structure introduced by features, resulting in a combinatorial explosion. Thus, using inheritance to complement object decomposition with functional decomposition negatively affects object decomposition.

[Clarke99] and [Tarr99] also explore design patterns in search of solutions to align the object decomposition with the functional decomposition. For instance, the Visitor pattern decouples functionality from the objects themselves by having a receiver method in each object and, for each functionality, a visitor class with a set of visitor methods[2]. Nevertheless, receiver methods still need to exist outside the features themselves, scattered throughout the class hierarchy. Adding new classes requires the creation of new visitor methods in each feature that is decomposed as a visitor. Furthermore, not every feature is prone to decoupling through the Visitor pattern. Other design patterns can help when the Visitor pattern is not applicable, but exhibit similar issues. For instance the Observer pattern, which is particularly useful for decoupling logging requirements, scatters notification invocations throughout the object decomposition. Other solutions using factories or proxies are also intrusive, requiring changes to the object types handled by existing code. More generally, solutions based on design-patterns require previous planning, for example by the introduction of hooks, visitor methods,

[2] There will be as many visitor methods in the visitor class of a particular functionality as the number of objects that are affected by it.

observer notifications or changing the object types used. Otherwise, introducing such elements latter on will involve an intrusive change.

[Clarke99] introduces design subjects as a functional decomposition for object oriented design. Each subject regards only a particular feature or non-functional requirement (like logging or security requirements) and exists both in design and code. Like in SOP, each subject is constructed by objects. So the functional decompositions introduced by subjects contain object decompositions. This way, subjects are not expected to limit the de-compositional power of object oriented design. The premise of enabling work to be done using existing languages and formalisms found in SOP is kept in SOD. [Clarke99] instantiates SOD for use with UML, in particular class and interaction diagrams. Each design subject contains a partial class diagram which will be composed with other subject's class diagrams to form a complete class model. Even though class diagrams are partial, each subject's class diagram will need to be complete in regards to the subject's functionality (i.e. the diagram may not reference functionality that it does not offer itself). This requirement is know as declarative completeness [Ossher99]. [Clarke99] gives an example of two features, each in its own subject, that use the same class hierarchy. Each subject needs to access data items about each class. As both features need these data items, they are replicated in the class diagram for each feature (subject). When it comes to implementation, declarative completeness may force each subject to hold replicated implementations. To avoid replicated code, [Ossher99] proposes for MDSoC the SOP equivalent to having only one subject actually implementing the replicated functionality while others only declare it (e.g. using abstract methods).

[Clarke99] introduces an additional composition construct: select composition. Select composition brings the novelty of being able to decide how to mix-and-match[3] features during runtime (based on runtime variables or configuration elements). Subjects composed using select composition can have a runtime variable determine which of them, or group thereof, will be chosen for execution. SOP only provides native support for static mix-and-match.

2.5 Conclusions

SOP retains OOP at its core but uses subjects as containers for different aspects of the same concept. It implements different kinds of subject composition, among which additive composition. In SOP, as a result of a process called flattening, existing class hierarchies are not retained after composition. SOP can also provide advanced composition features, like allowing the introduction of new features during runtime or being used to compose subjects written in different languages.

SOP brings functional decomposition to programming and, with it, additive change capabilities. SOD brings these to design. With SOD and SOP, direct traceability from requirements to code is achieved, even with design and programming formalisms that favour decomposition methods other than functional decomposition.

[3] Mix-and-match refers to the possibility of creating different flavours of a program by removing and introducing different features and components. In SOP, this is achieved statically with different subject compositions. Each subject composition can generate a different program.

Chapter 3

Multi-Dimensional Separation of Concerns (MDSoC) and Hyperspaces

Multi-Dimensional Separation of Concerns (MDSoC) was introduced in 1999 [Tarr99]. Like Subject Oriented Programming (SOP), MDSoC is a composition solution.

This chapter starts with a section justifying the need for MDSoC. This is done by acknowledging the domination of a particular decomposition criterion in each different formalism. One such case is the domination of the object-decomposition in OOP. MDSoC follows-up on the work done in SOP to address this issue. It extends SOP support for separation of concerns in the code artefact to a generalized model usable along multiple artefacts. MDSoC adopts a multi-dimensional structure for concern spaces that did not exist in SOP. In MDSoC, these structured concern spaces are called hyperspaces.

The MDSoC model is presented in Section 3.2, following [Tarr99] [Ossher99] and [Tarr01]. It is separated according to the two main stages in adopting the MDSoC model: decomposition and composition. Decomposition is achieved by populating a hyperspace which is structured by dimensions with the respective concerns. Composition is achieved by means of a separate composition definition: hypermodules.

Section 3.3 presents the Expression SEE example which is an example of MDSoC usage explored in most MDSoC literature, namely [Tarr99] [Ossher99] [Ossher00] and [Tarr01]. This example is also presented here as its implementation using Hyper/Net (our MDSoC implementation) will be detailed in Section 7.5.

3.1 The need for MDSoC

Section 2.4 addressed a relevant limitation commonly exhibited by most software artefact formalisms. Most of these only allow one dimension of decomposition, be it functional, object or any other. [Tarr99] calls this issue the “tyranny of the dominant decomposition” and considers it the major issue addressed by MDSoC.

Section 2.4 presented examples where the dominant decomposition of a particular artefact is used to support a different dimension of decomposition. One such example, in the code

artefact, using the OOP formalism, is that classes can be extended to separate their support of different features. In this example, the object dimension, that is dominant in OOP, is used for decomposition in the functional dimension. Although decomposition in a dimension other than the dominant one is achieved with these examples, the dominant decomposition is lost. In the previous example, the objects from the problem domain will end-up scattered in several sub-classes. Thus, even if the dominant decomposition of an artefact can be used for a different dimension of decomposition, it will only be usable for that dimension and looses its decomposition power in the dominant one. Furthermore the decomposition achieved in dimensions other than the dominant one is not at all optimized. Frequently, it is necessary to rely on artificialities: for example, the sub-classes from our previous example would identify features by a suffix in their name. As expected, some kind of alternative decomposition mechanism is required to achieve decomposition in more than one dimension with these formalisms.

Subject Oriented Design primarily addressed the issue of traceability from (functional) analysis to design and code. As such, Subject Oriented Design enhanced UML (an object oriented design formalism) to contain an additional decomposition along the functional dimension. This was achieved with the introduction of design subjects in UML. Subject Oriented Programming provided support for the same functional decomposition in the code artefact with subjects. But neither Subject Oriented Design nor Subject Oriented Programming were limited to adding only the functional dimension of decomposition to artefacts. In fact, several dimensions could be handled at the same time, given that appropriate subjects were introduced.

MDSoC is a follow-up to the work on Subject Oriented Programming by Harrison and Ossher. MDSoC supports multiple dimensions of decomposition in much the same way that Subject Oriented Programming and Subject Oriented Design do. That is, by separating concerns (Subject Oriented Programming and Subject Oriented Design did it using subjects[4]) and then providing a composition mechanism to compose the separated concerns into working blocks of software. In Subject Oriented Programming, subjects from all the dimensions will coexist at the same level. Thus, although SOP supports multiple-dimension decomposition, its model does not acknowledge this with any kind of structure for subjects, leaving them in an unorganized space. Additionally, SOP literature [Harrison93] [Ossher96] does not refer to multiple-dimensions of decomposition: it was MDSoC literature that acknowledged this concept [Ossher99] [Tarr99].

We have seen that SOP lacks an organization mechanism for its subject space. The semantics of composition impose the structure of this subject space, not allowing a structure based on different requirements. Furthermore, this structure only exists after a particular composition is done. There will still be no structure for the original subjects when using them in different compositions. MDSoC overcomes these limitations of SOP with a model for hyperspace that provides natural support for decomposition along multiple-dimensions.

SOP addressed only the code artefact in the OOP formalism. Subject Oriented Design had to be introduced to address the same needs at design level. MDSoC has a different approach. The MDSoC model was defined at a more abstract level, allowing it to be applied to all kinds of software artefacts and formalisms. MDSoC defines a hyperspace structure for concerns which is populated by elements from any artefact. This way, a particular concern will contain

[4] MDSoC concerns and SOP subjects can be seen as container elements for software units, where each container element represents a different perspective. The term “concern” was introduced with this meaning by Dijkstra [Dijkstra74]. [Dijkstra74] also introduced the concept of separation of concerns as a “technique for effective ordering of one’s thoughts”.

all relevant elements from the different artefacts. MDSoC refers to concerns instead of subjects, but these entities are pretty much equivalent; that is, concerns and subjects are both semantically significant containers. In the case of SOP, subjects can only contain elements from OOP code, while in MDSoC concerns can contain elements from any artefact formalism.

The MDSoC model works only as a complement to existing formalisms. MDSoC does not introduce any new formalism itself. This way, it should be easily used by anyone already acquainted with a particular set of formalisms. MDSoC can be used straight from the beginning of a software project until the end, being applied to the whole software lifecycle, or it can be applied at any point of the software lifecycle. In particular MDSoC can be used for refactoring existing programs, allowing an easier extension and evolution of these programs.

MDSoC shares with Subject Oriented Design the objective of fully supporting direct traceability between artefacts. MDSoC also aims to achieve a very high level of modularization flexibility to address complexity and provide comprehensibility for otherwise unintelligible software. MDSoC aims to promote reuse based on its advanced modularization capabilities, which also help to achieve a high level of decoupling. Finally, MDSoC should ease evolution by achieving a high level of decoupling, providing elevated comprehensibility and offering direct traceability between artefacts.

3.2 The MDSoC hyperspace model

MDSoC is centred on the MDSoC *hyperspace*, which is a multi-dimensional concern space. As can be seen in the previous section, concerns are a key element in MDSoC. Throughout [Tarr99] there are several references to different kinds of concerns such as functional requirements, customizations, concepts[5] (objects) and non-functional requirements. Many concerns will exist, belonging to each of these kinds. Features will usually map to functional requirement concerns, but some features might be achieved by offering customization concerns (on existing functional requirement concerns). Other features, like persistence and security, are non-functional, thus map to non-functional requirements. Non-functional requirements usually affect the entire functional concern set, thus they are overlapping. All of these different features will be related with real-life or virtual concepts or entities that make-up the problem domain, and form themselves one kind of concern. Each of the concerns belonging to the prior kinds may contain up to the entire set of concepts of the domain, thus overlapping with the concept concerns.

It can be observed that most of the different kinds of concerns overlap. This is one of the main issues addressed by MDSoC. In MDSoC, each of these concern kinds can be considered as a different dimension. Concerns define the coordinate system of each dimension, as can be see in Figure 1.

[5] It is important to stress that concerns and concepts are not the same thing. Concepts are representations of real-life or virtual objects while concerns are abstract grouping elements. At some points [Tarr99] uses the words concepts and concerns interchangeably.

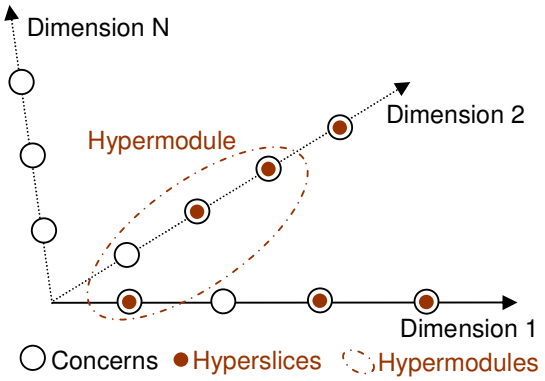

Figure 1. A representation of the MDSoC hyperspace.
Hyperslices contain the implementation units.
Only the implemented concerns are populated by hyperslices.

During the software lifecycle, concerns will be implemented using different artefacts. Initially, using requirement artefacts (like textual descriptions of requirements), then using design artefacts (eventually UML diagrams). Finally, the running implementation has to exist in a code artefact. But, more artefacts can be involved at this stage, namely test artefacts. The concerns are the same throughout these artefacts. But, as stated previously, most of the artefact formalisms only support one dimension of decomposition, called the dominant dimension of decomposition. Concerns belonging to any other dimension (or kind of concerns) will be scattered throughout the concerns of the dominant dimension of decomposition. Nevertheless, they will still exist in all artefacts, even if scattered.

Artefact formalisms are not monolithic but made up of parts. [Tarr99] calls these parts *units*. In MDSoC, different units may be placed under different concerns. While some units can be further divided, others cannot. Units that can be further divided are referred to as *compound units* while indivisible units are called *primitive units* [Tarr99]. In OOP, packages/namespaces and classes are compound units. Methods should also be considered compound units because they are made of statements. But, sometimes, indivisible units of an artefact are of a very low granularity. Supporting a hyperspace made-up of primitive units of a very thin granularity is harder than if compound units are chosen. Furthermore, very low levels of unit granularity may not be useful if units at a higher level of granularity always belong to the same concept. Back to OOP, even statements can be further divided. But, if we assume it would not be good practice to have the same statement address two or more different concerns, it would be a waste to consider statements as compound units. To cope with this, [Tarr99] defines that after a particular level of choice, compound units may be considered as primitive units: indivisible. As we will see, the original implementation of MDSoC for OOP – Hyper/J[6] – defines that methods are primitive units[7] [Tarr01]. Hyper/J does not allow decomposition beyond methods.

Without MDSoC we have the following decomposition hierarchy:

$$\text{Artefacts} \rightarrow \text{Modules} \rightarrow \text{Units}$$

There is always decomposition in the artefact dimension due to physical impositions (artefacts are separate). The modules of each artefact then provide decomposition in the dominant

[6] Hyper/J will be analyzed in some detail in section 5.1.
[7] For the sake of correctness, Tarr should have defined primitive units as the smallest units that can be composed.

dimension (for example, requirement specifications are decomposed functionally and in OOP classes provide decomposition in the concept/object dimension). MDSoC introduces three new top-level elements to this hierarchy:

Dimensions → Concerns → Hyperslices → Artefacts → Modules → Units

As seen previously, dimensions are grouping elements, holding concerns of the same kind. These concerns exist only conceptually and need to be instantiated as *hyperslices*. In MDSoC, it is common for a concern to be instantiated by a single hyperslice [Tarr99], but more than one hyperslice can exist for a single concern [Ossher99]. Hyperslices contain units from a particular concern. In case a concern is instantiated by a single hyperslice, as proposed in [Tarr99], all the units pertaining to that concern will belong to its respective hyperslice. These units may belong to different artefacts and can still be organized in the modules provided by the artefact formalisms inside the hyperslices. As such, each hyperslice may contain a part of the artefact dimension as well as parts of the dimensions in which each artefact formalism provides modularization. Both the artefact dimension and the dimensions of modularization of each formalism can exist as MDSoC dimensions.

3.2.1 Hyperslices

Concerns are abstract entities. To allow decomposition according to how units map to concerns, hyperslices had to be introduced [Tarr99]. Hyperslices provide the mean to implement concerns in particular artefact formalisms. Typically, each hyperslice corresponds and implements a particular concern. Thus, each hyperslice shares with its respective concern a place in a dimension of the MDSoC hyperspace (see Figure 1).

There are two distinct approaches to implement hyperslices in any given artefact formalism. The authors of the MDSoC model do not discuss these at an abstract level but we consider it an issue of importance.

One approach to creating hyperslices is to use the native decomposition features of each formalism. These are used to create modules that map a specific concern, thus creating the respective hyperslice. This approach can be referred to as the physical hyperslice implementation approach as it physically separates hyperslices. For example, in OOP, different features pertaining to the same object may be implemented in different classes. Each of these classes will then belong to a different package/namespace (a module in OOP). The same features for other objects may also be implemented in their own classes and share the same packages/namespaces. Each of these packages/namespaces is in fact a hyperslice populating our hyperspace. This example used the feature dimension, but other dimensions could be used. We have already focused cases of decomposition along dimensions other than the dominant one. In general, those decompositions work without any other kind of support, but invalidate decompositions in the dominant dimension. The kind of decomposition used to implement hyperslices is no exception, but we will see that MDSoC recovers lost dominant decompositions as a virtual dimension. The major advantage of the physical hyperslice implementation approach is that it allows the manipulation of hyperslices, by themselves, in a native module of a given formalism. For instance, a developer looking at a package/namespace that implements a particular hyperslice will only see units pertaining to the concern of that hyperslice.

The second approach is not as intrusive as the first one. Existing units will be decomposed along the dominant dimension of each formalism. Then a mapping system binds each of these

units to the appropriate concerns. This approach can be referred to as the virtual hyperslice implementation approach as, with this approach, hyperslices are not materialized and only exist virtually. Dimensions composed of hyperslices created this way may also be considered virtual. The virtual hyperslice implementation approach provides the only solution to map modules from different artefacts to the same hyperslice. It is also useful when existing software is decomposed, especially software for which no source code is available.

The physical approach has the great advantage of allowing the native manipulation of hyperslices, at least the parts belonging to each formalism. But the virtual approach offers important support both when more than one formalism is used in hyperspace and when existing software is decomposed. Thus, in practise, MDSoC uses a hybrid version of these two approaches. This hybrid version allows the physical separation of hyperslices in each formalism, but also offers mapping capabilities (inside the same formalism or in between formalisms). The hybrid version allows achieving the best physical decomposition, while having mapping capabilities to match modules in different formalisms to the same hyperslice. Furthermore, matching can also be used to match any physically indecomposable units to different concerns in different dimensions.

When either the physical approach or the hybrid approach is used, the dominant dimension of decomposition of each formalism will be lost. This dimension will only be usable in a native fashion locally in each hyperslice and in the result of composition. As such, for OOP, [Ossher99] introduces the object dimension, which Hyper/J generates automatically as the Class File dimension [Tarr01]. This is a virtual dimension, like the dimensions obtained from hyperslices created using the virtual approach. Still, an appropriate tool can provide adequate manipulation of units as seen from this dimension. The same process can be used to implement an object dimension for artefact formalisms lacking a native object decomposition mechanism.

The artefact dimension is not affected by any of the hyperslice implementation approaches. Simply because there is no way to physically modularize together units from different artefacts. Nevertheless [Ossher99] introduces the artefact dimension. The artefact dimension is physically supported by the fact that different artefacts are implemented in different formalisms.

In terms of the mathematical definition of a hyperspace (as a multi-dimensional coordinate space) hyperslices are considered hyperplanes [Tarr99]. This is because they contain units from different coordinates in the artefact dimension and in the dimensions in which each artefact formalism provides native modularization. Figure 2 depicts an example on a hyperplane defined by a particular concern (*Feature #1*) of a Features dimension.

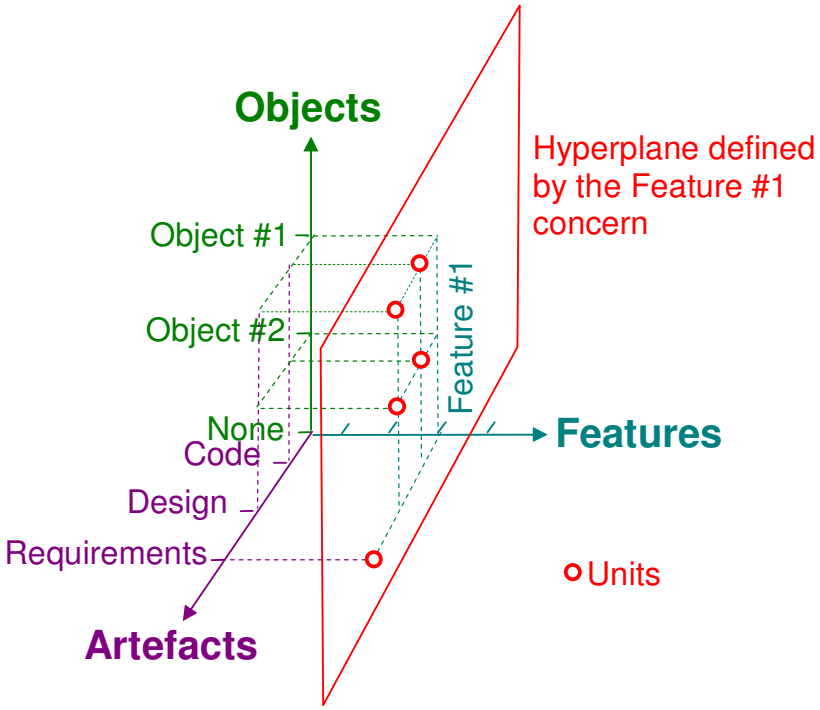

Figure 2. The hyperplane that is defined by a concern.

It is important to note that units can belong to more than one hyperslice. Hyperslices that share units are said to overlap. Sometimes overlapping hyperslices can be avoided by decomposing the shared units, so they become distinct, and relying on composition to join them again.

Hyperslices may depend upon units not contained there-in, thus depending on other hyperslices that provide these units. The same thing happened with subjects in SOP. To avoid this kind of coupling in MDSoC, [Ossher99] introduces the concept of declarative completeness for a hyperslice. Hyperslices have to be declaratively complete, that is, they must contain declarations for all referenced units not contained in them.

[Ossher99] adds that hyperslices in the same dimension never overlap, so one unit can exist only once in each dimension. This is an important restriction as it enables hyperslices/concerns to work as coordinates in each dimension of the MDSoC hyperspace. [Ossher99] also defines that each dimension must be complete, that is, all units in hyperspace must belong to all dimensions. This may be artificial for most situations. As such, [Ossher99] proposes the introduction of a special concern in each dimension – the None concern – holding all units that will not fit any of the other concerns in the dimension.

We have seen that dimensions can be defined based on the kinds of concerns emerging from a problem domain. Knowing when to add new dimensions to an existing hyperspace is also an important issue. [Ossher99] addresses it by realizing that hyperslices must fit in a dimension where they do not overlap any other hyperslices. If there is no such dimension for a particular hyperslice, either the hyperslice is incorrectly constructed or there exists no dimension adequate for this hyperslice. This last case is when a new dimension should be created. As an example, imagine we had the task of adding functional test methods to an OOP application

that belongs to a hyperspace with only one non-default[8] dimension: *Features*. Let us say these test methods all had to access internal state of existing objects to validate certain conditions. We would create a *Test* hyperslice with all the test methods required. But, we could not add the *Test* hyperslice to the Features dimension as it would overlap the hyperslices of features from which it accessed internal state. The *Test* hyperslice would not fit the Object dimension either. The Artefact dimension could be a choice, but we might want to consider tests as part of the programming artefact. If so, we had no choice but to create a new dimension to put our *Test* hyperslice in.

3.2.2 Hypermodules

Finally, after the hyperspace is decomposed, to create working blocks of software, a composition phase needs to take place. To execute this phase [Tarr99] introduces *hypermodules*. Hypermodules contain hyperslices (as depicted in Figure 1) and *composition rules* that dictate how to compose these hyperslices. These composition rules take units from different hyperslices in the hypermodule and from them create resulting units. The result of a hypermodule is then a collection of units. This result can be a complete piece of software, for example, a library, a service or a running program. The result can belong to a single artefact or span several artefacts. If, for example, the output of a hypermodule is a library, composition can generate a requirement document restricted to library functionality, design diagrams for the library, the library user documentation and the actual library code. But, the result of composition can also be incomplete, that is, it may contain references to units that it still does not contain, or it may fail to meet some other completeness constraint of the target formalism (or formalisms). In this case, the hypermodule can be used as any other hyperslice in new hypermodules. This is similar to how composed subjects can take the place of subjects in SOP composition.

MDSoC composition is based on SOP composition. It is a three step process [Tarr99]:

1. Matching: units inside concerns must be matched.

2. Reconciliation: any conflicts between units must be resolved.

3. Integration: the units must be transformed into a resulting unified unit.

[Harrison93] discusses different approaches of matching classes between subjects in SOP, some are automated matching mechanisms. SOP, as defined in [Harrison93], only supports the composition (and thus matching) of classes. In the light of MDSoC, we could consider that classes are the primitive units of SOP as defined in [Harrison93]. [Ossher96] extends SOP to support the matching of methods and variables. The MDSoC model allows matching units at any level. The only restriction to matching in the MDSoC model is that the units being matched must be of the same level (for example, methods match methods but not classes). Hyper/J supports matching units down to the method level, giving Hyper/J the same matching capabilities that [Ossher96] proposed for SOP. But, contrary to SOP, Hyper/J does not support any kind of non-explicit unit matching. Nevertheless, MDSoC poses no detriment to supporting non-explicit unit matching. Curiously, SOP does not provide any reconciliation mechanisms and Hyper/J does not implement any for MDSoC either. The need for reconciliation is identified in [Tarr99] but no instantiation is made in MDSoC literature.

[8] Here, we consider dimensions that are automatically created in MDSoC, like the object dimension or the artefact dimension, to be default dimensions. Similarly, the none concern can be considered a default concern.

Finally, regarding integration, [Harrison93] already provided some integration mechanisms for SOP: namely merge and override integration. Override integration simply discards all the units being composed but one, which is the resulting unit. Merge integration will create a resulting unit that transparently uses all the units that were composed. Merge integration can optionally compose a result value from the partial results of each composed unit. These two integration mechanisms are implemented in Hyper/J, among others [Tarr01], and will be focused in detail in Section 5.1.

[Ossher96] defines a formal model for SOP composition. As already stated, this model extends SOP composition from classes to methods and variables, in terms of matching and integration. Furthermore, [Ossher96] presents a rule model, already mentioned in Section 2.2. In this model, composition clauses express the matching and integration of units (classes, methods or variables) from different subjects. It would be possible to express all kinds of compositions in SOP only with composition clauses. But, frequently, the composition of classes, and sometimes entire subjects, uses the same composition clauses for all contained units (for example, match by name, integrate by merging). For this purpose [Ossher96] introduces composition rules that work as generators of composition clauses for units up to the entire subject level. A composition rule takes the place of as many composition clauses as there are matching units in the composed subjects. Composition clauses may still be used by themselves for expressing exceptions to the composition rule. [Tarr99] presents composition rules for MDSoC but does not define any lower level mechanism to express exceptions. A mechanism equivalent to SOP composition clauses, MDSoC composition relationships, is only defined in [Ossher99], within a formal model for MDSoC composition. The formal model in [Ossher99] is not extended with [Tarr99]'s composition rules. But, Hyper/J implements a model that is equivalent to the SOP model, mixing composition rules and composition relationships. Hyper/J relies primarily on composition rules and composition relationships can be used to express exceptions to composition rules. Each composition relationship, be it generated by a composition rule or expressed explicitly, identifies a set of units from the hyperspace (input units) and uses a composition function to transform these units into an output.

3.2.3 SOP and the MDSoC hyperspace

SOP and MDSoC share many similarities that have been described above. Nevertheless, SOP does not define any subject space like MDSoC defines the hyperspace for concerns. Even more importantly, SOP is specific to the code artefact in the OOP formalism and Subject Oriented Design is specific to the design artefact in the UML formalism. MDSoC defines a much more powerful model that can be applied to any artefact formalism and where artefact formalisms can coexist. Still the similarities are numerous, namely, the composition mechanisms of SOP and MDSoC are almost equivalent.

Another difference between SOP and MDSoC is that MDSoC, when implemented for OOP (like in Hyper/J), does not require any kind of binding object like SOP's object identifiers (*oids*). This is because MDSoC generates a composed result where a native entity (for example, a class) corresponds to the composed units. The composed result provides the operational "glue" that was provided by *oids* in SOP.

While supporting different overlapping class hierarchies between subjects, SOP introduced the concept of flattening to avoid issues like the diamond problem and others. Flattening removes all inheritance information from the result of composed subjects. MDSoC does not propose any flattening mechanism and Hyper/J does not implement any either. Without

flattening, it would be interesting to know how MDSoC addresses the issues with multiple-inheritance. [Ossher01] gives an example of the composition of two different overlapping hierarchies. As the example is composed using Hyper/J, and Java does not support multiple-inheritance, there should be some kind of mechanism to determine the resulting inheritance hierarchy. Nevertheless, neither [Ossher01] nor the Hyper/J manual [Tarr01] provide any information in this regard.

3.3 Using MDSoC: Examples

3.3.1 The Expression SEE example

Most of the MDSoC literature [Ossher99] [Tarr99] [Ossher00] [Tarr01] uses the same example to show how MDSoC should be applied to OOP and identify some of its inherent benefits. This example is presented as a Software Engineering Environment (SEE) with the respective requirements, design and code artefacts. The aim of the example is to represent expressions involving operations on variables and numbers. These expressions should then support several different features, such as being printed, evaluated and having their syntax and semantics checked.

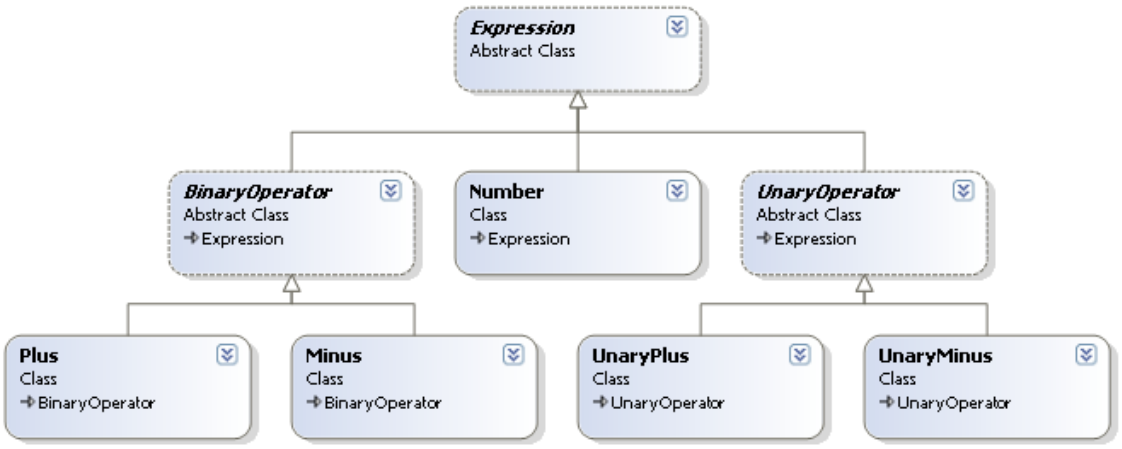

Figure 3. Class hierarchy used in the Expression example.

At first, while presenting this example, MDSoC literature describes how it can be implemented in OOP without using MDSoC. Each expression component is a class, creating a class hierarchy with an *Expression* class as its root element (see Figure 3). For instance, there is a *Number* class that derives directly from the *Expression* class. *Minus* and *Plus* operator classes derive from a *BinaryOperator* class which then derives from *Expression*. Finally, there is a *UnaryOperator* class, also deriving from *Expression*, which is extended by *UnaryMinus* and *UnaryPlus* classes. This class hierarchy is common to [Ossher99], [Tarr99], [Ossher00] and [Tarr01]. Only with operator classes and a number/literal class, it provides no

support for variables. Nevertheless, this support is implemented, using a *Variable* class, in the demonstration code for this example that comes with Hyper/J[9].

Each feature is then implemented using a different method that needs to exist in most, if not all, of the defined classes. To evaluate expressions, an *eval* method, that returns the computed value for the expression, can be used. Expressions can be printed out using a *display* method. To check expressions there is also a *check* method that returns true if the expression is valid.

There is no clear separation for the implementation of each feature. This way, removing a particular feature is not trivial. Introducing new features may require adding new methods to existing classes, but might also be more complex. [Tarr99] proposes the introduction of a persistence feature along with a style checker. Implementing persistence requires changing the constructor and properties of the Expression class hierarchy objects so these write any changes to their values to a persisted store. Applying design patterns to achieve a more decomposed implementation of this feature is possible but suffers from drawbacks described in Section 2.4.

The style checking feature, also proposed in [Osher00] and [Tarr01], could be implemented using a new method. Still, [Tarr99] discusses how it would be possible to execute all the checks (syntax, semantic and style) through a single method. To achieve this [Tarr99] proposes the usage of a visitor design pattern for each kind of check. Like using design patterns for the persistence feature, this also has some of the drawbacks already discussed in Section 2.4. Eventually, a more adequate solution would be to implement a single *check* method in the Expression class. This method could invoke the desired different check methods and it would be a single point of change to obtain different combinations of the different types of checking. As this method is inherited by all other classes, invoking it for a particular class executes the different check types for that class. This solution is fully supported by inheritance but was not addressed by [Tarr99] or the remaining MDSoC literature which proposes a style check feature [Osher00] [Tarr01].

[Osher00] and [Tarr01] additionally propose the implementation of logging. This involves writing relevant information like entering or leaving a particular method, its arguments and return value. This non functional feature requires adding logging statements to all the methods in all the classes. Alternatively, an observer design pattern can be used, but it requires the introduction of notification invocations at the same points. This is yet another drawback of using design patterns that was already presented in Section 2.4.

[Ossher99] proposes the introduction of a caching feature, to avoid having to re-evaluate the same expressions. A cached result variable can be introduced. When a first evaluation occurs this variable should be written. If an evaluation is requested and there is a cached result it should be directly used. This behaviour has to be implemented in the *eval* method, mixing the caching feature with the existing evaluation feature. Furthermore, like persistence, the caching feature requires changing class properties and constructors to invalidate a cached result whenever expression components are changed.

The elements of MDSoC literature presenting the Expression SEE example go on to present how it is implemented using MDSoC and how this makes it much easier to extend with new features. They define a hyperspace with two dimensions. The first is the Object dimension and is already defined by the class hierarchy of the non MDSoC solution. Each class corresponds to one concern in this dimension. The second dimension is the Features

[9] A downloadable Hyper/J package that includes this example is available at http://www.alphaworks.ibm.com/tech/hyperj/download.

dimension. It is populated by concerns pertaining to the different features like display and evaluation. The Features dimension can be physically implemented or be created using virtual decomposition. It can even contain both physically and virtually decomposed units.

[Tarr99] provides a physically decomposed model for the Features dimension. It uses only units from the design artefact as it does not address the code artefact directly. So, [Tarr99] presents class diagrams where the class hierarchy is replicated for each concern. The classes in the diagram for a particular concern contain only the members that address that concern. For instance, the classes in the Evaluation concern only contain the *eval* method whereas the classes in the Display concern only contain the *display* method. In some concerns the class hierarchy is not complete in regards to the original one because some classes are not relevant for that concern. This happens in the Evaluation concern where there is no general evaluation for unary or binary operators, so their children (*Minus*, *Plus*, *UnaryMinus* and *UnaryPlus*) derive directly from *Expression*.

On the other hand, [Ossher00] and [Tarr01] focus on the code artefact and, using Hyper/J, implement most of the concerns in the Features dimension virtually. This is done by identifying the concerns to which the units belong. For instance, the *eval* methods in all the classes belong to the Evaluation concern. Similarly the *display* methods belong to the Display concern. These methods continue to exist in the original classes but are virtually mapped to these concerns in the Features dimension. This allows using the original OOP code in an MDSoC hyperspace, without changing it, but benefiting from MDSoC advantages as we will see further on.

A third dimension, the artefact dimension, is explicitly addressed in [Ossher99]. Nevertheless, it is natively present in all Expression SEE descriptions simply because they contain separate requirements, design and code artefacts.

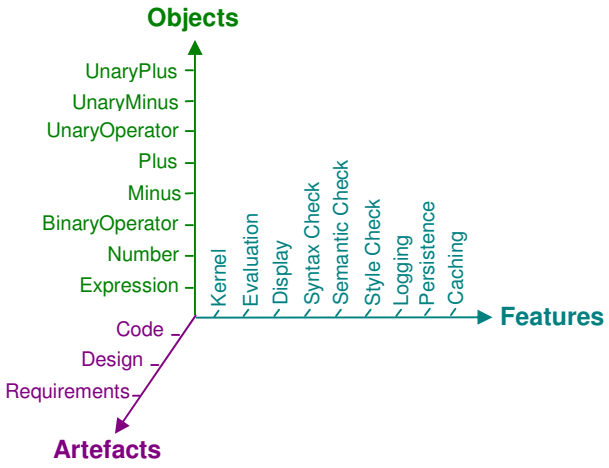

Figure 4. Representation of the MDSoC hyperspace used for the Expression SEE example.

Figure 4 represents the three dimensions used in the MDSoC hyperspace for the Expression SEE. Each dimension is represented with its own colour and indexed by its own concerns. As seen previously, this hyperspace is populated by code, design and requirement units using either virtual or physical decomposition.

Back to the Features dimension, the Kernel concern is a common concern in many MDSoC hyperspaces. It provides basic constructors, properties and variables for all the classes involved. Other concerns may provide other constructors, properties and variables that are

particular to those concerns. The Kernel concern is present in all of the versions of this example in [Ossher99], [Tarr99], [Ossher00] and [Tarr01].

As for checking concerns, [Tarr99] separates syntactic from semantic checking in different concerns, while [Ossher00] [Tarr01] has a single Check concern that addresses both types of check. In this last case, the Check concern contains a *check* method for all the classes in the hierarchy. In the case of [Tarr99], MDSoC composition is used to merge the *check* method from each checking concern. The merge is done so that the resulting method only returns true if both syntactic and semantic checks return true.

One of the advantages of MDSoC presented with this example is the ability to mix-and-match different features creating different versions of the same application. For instance, with the hyperspace defined in [Tarr99], it is possible to remove either syntactic or semantic checking by removing the respective concern (which is a trivial task).

Adding style checking to this example is also trivial. Style checking introduces its own concern in the Features dimension. The classes in this concern provide their own *check* method. Then, MDSoC merge composition is used to compose this concern with the other check concerns. In [Ossher00] and [Tarr01] this concern is implemented using a physical decomposition. Physical decomposition is usually better as it allows the programmer to work with decomposed code. The original feature concerns (Kernel, Evaluation, Display and Check) were not physically decomposed in [Ossher00] and [Tarr01] to avoid having to change the original OOP code, which was developed without MDSoC support.

The caching feature [Ossher99] is implemented in MDSoC as a separate concern. As identified previously, the caching feature affects the *eval* method in all classes in the hierarchy and also their constructors and properties. This means the Caching concern overlaps the Kernel and Evaluation concerns in the Features dimension. It also overlaps all concerns in the Object dimension. When this happens with a new concern, it should be introduced into a new dimension. [Ossher99] proposes the introduction of a Caching dimension. We consider that a more general Non-functional requirements dimension might be more adequate, being prone to receive other related concerns. The Caching concern should contain constructors, properties and an *eval* method for each class in the hierarchy. These members only need to update or invalidate a local evaluation result variable. This Caching concern should then be used in a hypermodule that defines how it is composed with the evaluation and Kernel concerns. [Ossher99] does not provide any details about the composition of this concern. But, for instance, the constructors and properties can simply be merged with the ones in the Kernel concern. This concern was represented as part of the Features dimension in Figure 4 to avoid representing a four dimensional hyperspace. Placing this concern in the Features dimension may not be the most adequate approach but was followed for other non-functional requirements (persistence and logging) in [Tarr99], [Osher00] and [Tarr01].

[Tarr99] also proposes the implementation of persistence as a separate concern. It does not detail a particular solution for this feature, not even at design level. Yet, this concern is similar to the Caching concern. It also overlaps the Kernel concern and, optionally[10], others, which provide state information that can be persisted, for instance, the check and even the Evaluation concern. As such, this concern should also be implemented in a different dimension. The Non-functional requirements dimension we proposed for the Caching concern is a good candidate. All the class members that are used to change or compute state

[10] It is discussable whether persisting state information that can be computed is a task for the persistence concern or the caching concern. [Tarr99] seems to defend it is a task for the persistence concern by proposing that it also persists the result of the *check* method.

information about expressions should be implemented in this concern. These members should simply save the new state information without executing any other tasks, like initializing variables or computing state information. Then, this concern should be introduced in a hypermodule, defining an adequate composition with the kernel and other concerns.

The description of how to implement the logging feature with MDSoC [Osher00] [Tarr01] is more detailed than the description of the previous features. A specific Logging concern is also used. Logging method calls, with the respective entry and exit, affects the methods in all existing concerns. This could mean that the logging feature overlaps all other concerns and should be implemented in its own dimension. Nevertheless, in [Osher00] and [Tarr01], the Features dimension is used for the Logging concern. It would be more adequate to create this concern in the Non-functional requirements dimension.

The Logging concern is implemented using the observer design-pattern. A call to a method entry handler is embedded into the beginning of existing method bodies. A similar call is embedded into the end of the same method bodies. Embedding these method handler calls is done using a new composition function implemented in Hyper/J: bracket. Bracket allows the introduction of method invocations before and after other methods (for more details see Subsection 5.1.2). The handlers simply write relevant method information to a log file. The Logging concern should be used in hypermodules that define the composition of these handlers with all the methods that need to be logged.

3.3.2 Other examples

[Ossher01] presents a different but similar example to the Expression SEE. The example consists of an employee class hierarchy that provides functionality (methods) that address two main features. The first is a personnel feature that keeps track of employee details. The second is a payroll feature and is centred around a *pay* method. Additionally, each feature has related business rules it needs to enforce. The example is implemented using MDSoC in a similar way to the expression example. The two different requirement groups are implemented in different concerns of a Features dimension. The Personnel concern is similar to the Kernel concern in the expression example and contains constructors and properties for the employee classes. Additionally, a Business Rules dimension contains concerns that implement particular business rules, each in its own concern. This separates the business rules from the features they are related to.

Both the expression and employee examples consist of a single class hierarchy where all the classes share a common set of methods. Each method offers the functionality of a particular feature or accumulates functionality from different features. Each method can also be affected by non-functional requirements. It can be argued that these examples are too specific and, by sharing the characteristics presented, do not capture a wide range of application types. For instance, the examples do not capture applications containing several different class hierarchies which are not interrelated by inheritance but by reference. Nevertheless, this does not mean that different application types can not equally benefit from MDSoC. It only means that different examples of using MDSoC should be studied to achieve a wider validation ground for the benefits of MDSoC.

There are also real-world examples of MDSoC usage. One such example was the implementation of the GNU sort application using Hyper/J [Carver02]. MDSoC has also been applied to several fields in computing, namely web service composition [Hailpern01] [Arsanjani03] and middleware [Rouvellou00]. As for other fields, it has been used in the

design of hardware embedded systems [Stuikys02], in music composition [Hill06], to cope with the complexity of regulatory text bodies [Lasky03] and in sustainable architecture and urban planning [Lourenci02]. [Lourenci02] provides a particularly interesting perspective on MDSoC.

3.4 Conclusions

MDSoC is an evolution of SOP that has acknowledged the multiple dimensions in which subjects (or concerns) can be organized. In MDSoC, these dimensions are supported by a structure called hyperspace. Hyperspace is made-up of concerns that are contained in dimensions. Concerns contain units from different software artefacts and/or formalisms. This makes MDSoC a cross-artefact model. MDSoC is also based on two stages: decomposition and composition. In the first stage, units are placed, physically or virtually, inside concerns of the hyperspace. In the second, sets of units are matched and composed according to composition rules and relationships. Like SOP, MDSoC adopts composition strategies defined in composition rules and allows expressing as many exceptions to these as required.

Finally, we used the Expression SEE example to demonstrate a set of features that severely benefit from being implemented using MDSoC.

Chapter 4

Technological background

This brief chapter presents an overview of the technological background required for the chapters that follow. First, the Java programming language is very briefly addressed. This will help readers not acquainted with Java cope with Section 5.1, about Hyper/J, which is an MDSoC implementation for Java.

The core of this chapter is a slightly more detailed description of the Microsoft .NET Framework and its languages. It then focuses on .NET partial types, which is an important feature in regards to MDSoC, as Section 6.1 will show.

Hyper/Net, the MDSoC implementation developed to help support the thesis documented herein, adds MDSoC support to Microsoft .NET languages. Hyper/Net was integrated with two .NET Interactive Development Environments (IDEs), SharpDevelop and Microsoft Visual Studio. These are also addressed here.

This background information on Microsoft .NET will also help with Section 5.2, on HyperC#.

4.1 The Java programming language

Java is an OOP language based on C and C++. Its first version (1.0) was released in 1995 by Sun Microsystems. Java did not adopt most C and C++ low-level facilities, namely pointer manipulation. With its minimalist design, Java also does not offer support for other common OOP features, such as multiple-inheritance [WikiJa].

Java programs can use a set of common functionality from standardized Java libraries. These provide access to system functionality like user interfaces, networking, etc. One interesting point is that these libraries are equally available in all platforms that support Java programs. In fact, Java programs are not compiled into native machine code but into an intermediate language: Java bytecode. Java bytecode programs are usually compiled into native code at runtime, as needed, by a Java Virtual Machine. There are Java Virtual Machines for almost all platforms. This makes Java programs portable across platforms.

Java was originally closed source, but was freely available for use. As of May 2007, Sun Microsystems publicly released Java's compilers, virtual machine and other tools, officially making Java an open-source, community project.

4.2 Microsoft .NET Framework

The first version of the .NET Framework was released in early 2002 [WikiNF]. It was introduced with the intention of providing a common framework for Microsoft's programming language implementations.

An important part of the .NET Framework is its Base Class Library (BCL). It provides common functionality in areas like data access, user interfaces and networking, among others. Programs developed in any .NET language can use the BCL. In fact, it is available in the same way from all .NET languages.

The .NET Framework also manages the execution of programs written for the framework. This support is offered, usually at runtime, using virtual machine technology. The .NET Framework includes a virtual machine that translates a .NET intermediate language (the Common Intermediate Language – CIL) into the machine language of each platform (see Figure 5). The resulting machine code is what ends up being executed. Like the Java Virtual Machine, this provides independence from the architecture of each platform that .NET programs run on.

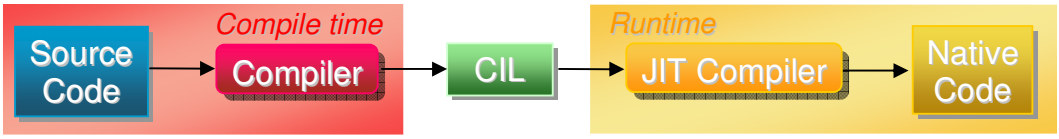

Figure 5. Typical .NET compilation and runtime process.

Like Java, the .NET Framework also supports an intermediate byte-code language (the CIL) using a virtual machine model. Also like Java, the .NET Framework offers a set of common functionality through system libraries, like the Base Class Library (BCL) [WikiNF]. Yet, the .NET Framework takes these advantages one step further than Java by supporting .NET in multiple languages. Each .NET language needs only to have a CIL compiler that translates it to the CIL. This is of crucial importance for the adoption of the framework. Any existing language, in particular OOP languages, can be ported to a .NET variant. An immediate advantage is having direct access to system functionality through the BCL, allowing language implementers to concentrate on language features instead of system integration features. At the time of writing, .NET supports more than 30 different programming languages, most of which were developed by parties other than Microsoft [WikiNL]. In comparison to Java, the .NET Framework falls short on portability, as Microsoft only offers an implementation for Microsoft Windows platforms. There are open-source efforts (like Mono) to provide cross-platform support for the .NET Framework, but .NET support is not complete, namely supporting only part of the BCL [WikiNF].

The .NET Common Intermediate Language (CIL) is object oriented. It natively supports features such as class inheritance and polymorphism [Thai03]. It also instantiates a particular set of language entities from common OOP concepts:

- Classes – can be considered the most important entity in the CIL. All other programming elements must belong to classes, with the exception of interfaces.

- Interfaces – declare a set of member signatures (methods, properties, etc.) that must be implemented by classes that derive from the interface.

- Methods – are the main implementation point for functionality. Methods must belong to classes and can be declared in interfaces. A method is identified by its signature: its name; set of parameters types and return value type.

- Fields – represent class member variables.

- Properties – are similar to fields but provide greater encapsulation by being accessed through getter and setter methods.

- Events – provide a native implementation of a synchronous observer design-pattern.

- Namespaces – serve as a modularization tool, separating different types inside a CIL module according to any desired criteria.

The CIL language entities are enhanced with another set of standardized language entities. These are defined as part of the .NET Common Type System (CTS) and can be mapped to CIL objects [Ecma02]. They are:

- Exceptions – which are particular classes (deriving from *System.Exception*) that can be used when errors occur.

- Enumerations – which are special types that contain a set of named constants.

- Indexers – which are operators that allow accessing objects as if they were arrays.

- Delegates – which are a type-safe version of the function pointers from other languages, like C. Delegates define a method signature type[11] which can be used to declare invocable variables. These variables can be associated with any methods with the appropriate signature [Thai03].

- Attributes – which are particular classes (deriving from *System.Attribute*) that can be applied to most language elements to provide metadata on them. Some attributes have a specific semantic value for the CIL and are provided with the .NET BCL. Custom attributes can be created by extending the *System.Attribute* class [Liberty01].

Exceptions are already class types at CTS level, and so should also be classes in .NET languages. Enumerations and delegates are converted into class types for the CIL.

Delegates and exceptions are class types in the CIL. Nevertheless, the CIL validates delegates according to particular rules and needs to provide native support for exception handling [Ecma02]. Additional information about these particular types needs to be provided to the CIL in the form of metadata. Attributes are also introduced into this body of metadata.

All .NET languages must implement the .NET Common Type System (CTS). It is noticeable that the CIL and the CTS strongly influenced the design of the C# and VB.NET languages.

[11] Here the method signature is defined without the method name.

The entities that exist in both languages are exactly the ones supported by the CTS. Each of these .NET languages is focused in slightly more detail in the two following Subsections.

4.2.1 The C# programming language

C# is an OOP language developed by Microsoft and approved as a standard by the ECMA and ISO organizations [WikiCS]. The first version of C# (C# 1.0) appeared in 2001, just before the official release of the first version of the .NET Framework. C# is based on C++ but also includes elements from Delphi and Java. The most relevant C# design goal is simplicity. Like Java, C# was chosen not support multiple inheritance. C# implements a unified type system rooted in the *Object* class. Unlike Java, in C#, primitive types, like integers, booleans and so on, are also part of this type system by extending the *Object* class. The unified type system is not particular to C# but is a consequence of the .NET Common Type System (CTS).

4.2.2 The VB.NET programming language

The Visual Basic .NET, or VB.NET, programming language is an evolution of Microsoft's Visual Basic to support the .NET Framework. Visual Basic is an event driven programming language. It is used mainly for rapid application development (RAD) of user interfaces and reusable components (COM objects) in the Windows platform [WikiVB]. VB.NET retained part of the Visual Basic syntax and the event driven capabilities, using .NET Framework delegates and events, but became a full-fledged OOP language equivalent to C#.

4.2.3 Partial Types

As part of a set of new language features, Microsoft introduced partial types with the C# and VB.NET 2.0 language definitions in 2005 [CSharp05]. Partial types use a type modifier (*partial*) that enables separating type definitions throughout as many files as desired. Listing 1 and Listing 2 exemplify the usage of partial types to implement two different methods (*Eat* and *Sleep*) for the same class (*Fish*) in separate files. When the two listings are compiled, there will be a single *Fish* class providing the two implemented methods, as if they had been defined inside a single class in one file.

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

Listing 1. Declaration of the *Fish* partial class in the first file, using C#.

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

Listing 2. Declaration of the *Fish* partial class in the second file, using C#.

Partial types allow defining different members of the same type separately, eventually in different files. The *partial* modifier can only be applied to classes, structs and interfaces. It is not valid for delegates or enumerations [CSharp05].

Currently, the most common usage for this feature is separating tool-generated code from human-generated code for the same class. By generating classes with a partial modifier, members can manually be added to them in separate files. This enables each class to be regenerated using the tool without overwriting the human-generated portion of the class [CSharp05]. This is particularly useful when extending the tool-generated class is not a valid option. [CSharp05] also proposes the usage of partial types to allow different programmers to work on the same type separately.

[CSharp05] goes on to provide important implementation details that determine how partial types can be used:

- The partial types for a particular type must all be defined under the same namespace. Two partial types with the same name, defined in different namespaces, are defining different types, each belonging to its respective namespace.

- All the parts of a type that uses partial types must be declared partial and must have the same accessibility; for instance, all are public or all are private.

- If any of the partial types of a particular type is defined as abstract, the resulting type will be abstract.

- Each partial class can extend another class. Still, all of the different partial classes, for a particular class type, that extend a class, must extend the same class. If at least one partial class extends another class, the resulting type will also extend that class.

- Partial types are composed into a type (by a .NET compiler) in an additive fashion:

    - All the members defined in each partial type with exist in the resulting type.

    - Any interfaces implemented by a partial class or extended by a partial interface will also be implemented or extended by the resulting class or interface (respectively).

    - Any class attributes that are applied to a partial type will also be applied to the resulting type.

    - Any XML comments that applied to a partial type will also be applied to the resulting type.

- All partial types of a particular type must belong to the same assembly and module.

Even though .NET, in particular the C# language, has adopted many of its language features from Java, there is no equivalent to partial types in Java.

As we will see, from Chapter 6 onwards, partial types can be used to achieve separation of concerns. Nevertheless, references about this usage for partial types are scarce. [Hirschfeld03] briefly discusses how partial types are expressively more limited than Aspect Oriented Programming (another composition approach) for addressing separation of concerns.

4.2.4 SharpDevelop

SharpDevelop is an Integrated Development Environment (IDE) for .NET. The task of an IDE is to streamline the process of programming. By also offering support for related tasks like designing, testing or documenting, IDEs tend to be the tool of choice for streamlining the entire software development process.

The SharpDevelop IDE supports the two main .NET programming languages, C# and VB.NET. It also supports the Boo programming language which is an OOP language based on Python. SharpDevelop itself is developed using C# and runs on the .NET Runtime. It is an open-source project and is free to use.

The first chapter of [Holm03] provides an overview of the SharpDevelop functionalities. Some of that information is outdated and had to be reconciled with updated data from the SharpDevelop website [SharpDevelop07].

SharpDevelop provides common source code editing features like search and replace and syntax highlighting. Dynamically, while users are writing code, SharpDevelop provides code completion suggestions based on the source code context. For instance, while starting to write the name of a method invoked from a class, SharpDevelop provides a list of all methods in the class that start with the characters already written. The programmer does not have to finish writing the method name and can choose it from the list. Another SharpDevelop feature that is related with code completion is called method insight. While the programmer writes the invocation of a particular method, SharpDevelop provides information about the required arguments, highlighting the current argument while it is typed. SharpDevelop also supports integrated debugging. It allows breakpoints to be defined in the source code and uses them in interactive debug sessions.

Another important concept managed by IDEs is the concept of programming projects. Projects act as source code aggregators. They contain related source code files that should be compiled together, along with other elements, like resources: images, localized text strings, etc. Projects also simplify the source code build process, allowing an entire project to be built with a single click. Any errors during the build process can be presented to the programmer inside the IDE and he/she can navigate to their source. SharpDevelop fully supports several different kinds of project files, namely the MSBuild project files used by Visual Studio, which is the main, commercial, .NET IDE.

A particularly interesting feature of SharpDevelop is that it provides the possibility of translating programs in any supported language into any other supported language. This feature is supported by a parser that is part of NRefactory, which is addressed further on.

Another area where IDEs tend to provide solid support is on Rapid Application Development (RAD) for interfaces. SharpDevelop provides visual tools to create Windows Forms and Web Application (ASP.NET) interfaces.

Finally, SharpDevelop provides important extension and integration features that allow more functionality to be plugged-in through separate modules. Such integration features are used to plug-in external tools like, testing and code-coverage frameworks (NUnit and NCover) or documentation tools (like NDoc). Several other plug-ins exist and many are under development.

NRefactory

Despite the name, NRefactory is a C# and VB.NET source code parser. It is used internally by SharpDevelop, mainly for source code translation. After parsing source code, NRefactory allows the manipulation of its Abstract Syntax Tree (AST), providing a programming interface for manipulating code [NRefactory05]. The contents of the AST can be converted back into textual source code, in any of the two supported languages. NRefactory only supports processing a single source code file at a time.

4.2.5 Microsoft Visual Studio

Microsoft Visual Studio is the first and best known IDE for the .NET Framework. It is a commercial product developed by Microsoft and is closely related with the development of the .NET Framework. Usually, the IDE support for new .NET Framework versions is simultaneously made available in the form of new Visual Studio versions or updates.

Visual Studio heavily inspired SharpDevelop. Most of its features were based on similar ones that are available in Visual Studio. Since its origins, Visual Studio offers Intellisense, which is one of the first code completion features to be available in an IDE [WikiIS]. Visual Studio 2005 introduced its own test projects, offering unit testing facilities that are closely based on NUnit. Another feature of Visual Studio 2005 is the Class Designer. It offers the ability to create class diagrams from existing code or from scratch. The most interesting aspect of the Class Designer is that class diagrams are dynamically synchronized with the source code. Changes made in the class diagram take immediate effect in the source code and vice-versa [Stoecker04]. This way, class diagrams not only serve to represent code, but provide a higher level code interaction mechanism. We use Visual Studio generated class diagrams extensively throughout this document.

Visual Studio has its own XML project format (MSBuild) and supports several different types of projects. Each type of project is adequate for a particular architectural role. For instance, there are ASP.NET web application projects that can expose user interface functionality through a web interface and need to be supported by a web server. Windows forms projects allow creating rich Windows client applications. Simpler project types are also supported. Console application projects can be used to develop command line applications which offer very limited used interaction facilities. Class library projects allow implementing software modules offering functionality that can be used from other projects. Most medium to large projects need to rely on several class library projects. Visual Studio project types are simply templates that configure .NET projects and introduce sample code in a way that is appropriate for each scenario. Most Visual Studio project types are similarly supported in SharpDevelop.

4.3 Conclusions

Java and the .NET Framework have a lot in common. Both are centred on OOP languages and concepts, provide vast libraries for access to system functionality and implement a cross-platform virtual machine environment. One .NET feature that does not exist in Java is .NET partial types. .NET partial types allow scattering a class definition throughout as many different files as required. Another important element in the adoption of both platforms is how they are supported in IDEs that streamline the developers work. With a powerful set of

development support and guidance tools, IDEs tend to be used during the entire development process. Some of these tools and features, like project files, code completion and testing tools, will be revisited further on. We will also be revisiting NRefactory, which is a C# and VB.Net source code parser.

Chapter 5

MDSoC implementations

The conceptual model of MDSoC is very promising as analysed in Chapter 3. Yet, without implementations it would be nothing more than a model. This chapter presents two existing MDSoC implementations that can be used by developers. Hyper/J is the first MDSoC implementation and works with the Java language. HyperC# is a more recent project that can be used with the C# language. Each section of this chapter focuses one of these implementations. It presents their features, usage scenarios, bridges them to the MDSoC model and summarizes their limitations. Another MDSoC implementation, developed by us, is called Hyper/Net and is presented in Chapter 6.

As noted, the MDSoC implementations presented here only address the code artefact. Curiously, there are many other MDSoC implementations and processes that address the design artefact [Herrmann00] [Memmert02] [Philippow03] [France03], the architecture artefact [Kande00] [Kande03] and the analysis artefact [Sutton02] [González05]. These are not analysed here as our aim with this chapter is to provide comparison grounds for Hyper/Net, which does not address design or analysis artefacts. A comparison between the three MDSoC implementations for the code artefact (Hyper/J, HyperC# and Hyper/Net) is done in Section 9.2.

5.1 Hyper/J

Hyper/J instantiates the MDSoC model for the Java language. It is the first implementation of MDSoC and was developed by the authors of the MDSoC model [Tarr01]. Hyper/J works with Java class files as input and also outputs Java class files [Ossher00]. It is a post-compilation, pre-runtime composition tool.

Along with input class files, Hyper/J uses three different metadata files [Ossher00] [Tarr01]. One of the files defines which units from input class files are used in hyperspace (project specification). Another defines dimensions and their concerns, and how the units in hyperspace populate these concerns (concern mapping). Finally, hypermodules are also defined in their own files. Hypermodule files identify the set of concerns/hyperslices used and how these are composed using composition rules. Once created, these metadata files can be reused in new compositions. For instance, to create a different hypermodule for the same

hyperspace, the project specification and concern mapping files can be reused with a new hypermodule file.

Java developers may or may not have in mind that their code will be used in a particular hyperspace. Hyper/J supports both scenarios. Furthermore, it can even be used when there is no source code available at all, as it works directly with class files. In all situations, Hyper/J requires no intrusive changes to existing code to place it in a hyperspace and then use it in compositions [Ossher00].

Hyper/J uses the hyperspace definition along with a hypermodule to create output class files. These output class files are the result of Hyper/J composition and can be reused in new compositions or, if complete, as a running program.

Hyper/J implements MDSoC limited to a single artefact (code) and a single formalism of this artefact: the Java language.

Packages, interfaces, classes, their members (like methods and fields) and statements are Java language units. Hyper/J offers composition constructs for all of these units except for statements [Tarr01]. Decomposing class members into statements is pointless as Hyper/J offers no means to compose the resulting statements into class members again. Yet, class members can usually be decomposed into sets of equivalent, usually smaller, class members. For instance, a method can be decomposed into methods with the same signature, each holding only part of the original method body. Contrary to decomposition into statements, decomposition into equivalent class member types is useful, as Hyper/J offers composition constructs for class members. This way, class members are the smallest decomposable units but, at the same time, also primitive units of a Hyper/J hyperspace. Class members are primitive units because they are the smallest units resulting from decompositions for use with Hyper/J. This makes packages, interfaces and classes compound units.

Class members in this approach can be seen as a particular case of a wider scenario where the decomposition of a unit always results in a unit of the same type. If this kind of unit is not further decomposable into smaller granularity units, it will still be considered a primitive unit, even though it is decomposable. This can be considered as an exception to the definition of primitive units from Section 3.2.

5.1.1 Dimensions, concerns and hyperslices

Through the project specification file, Hyper/J allows the declaration of all the units that are included in each hyperspace [Tarr01] (Subsection 4.2.1). These files contain directives which identify specific classes or interfaces. Some directives identify single classes or interfaces. Others are more powerful and allow the inclusion of the entire set of classes and interfaces inside a package or a file-system directory. Pattern matching can be used to filter the included units. In Hyper/J, not all classes and interfaces included in a hyperspace are composable. The directives in the project specification file also allow defining which classes and interfaces are composable and which are not. Any classes that are referenced by classes introduced into hyperspace are automatically introduced into hyperspace by Hyper/J. Yet, these are defined as not composable by default. If required, introducing these referenced classes explicitly allows them to be composable.

Compound units other than packages must be introduced whole into hyperspace. Primitive units can only be introduced into hyperspace as part of a compound unit. For instance, a class

member cannot be introduced into hyperspace by itself. Its entire containing compound unit (the class) must be introduced. It is not possible to define that part of a compound unit cannot be used for composition while the remaining can. To introduce a particular compound unit member without introducing the remaining members of the unit, it must be decomposed before being introduced into hyperspace. A new compound unit should be created with only the members that are to be introduced into hyperspace. Of course this decomposition is subject to the completeness constraints that allow the compilation of the compound units involved. This issue is further discussed ahead in this section.

After units are introduced into hyperspace they must be matched to the adequate concerns. Hyper/J uses the concern mapping file to declare hyperspace dimensions with concerns and match units to these concerns [Tarr01] (Subsection 4.2.2). Hyper/J implements the most limited hyperslice model where hyperslices are equivalent to concerns. This model was proposed in [Tarr99]. As we will see, this is also the model used in the other MDSoC implementations (HyperC#, see Section 5.2, and Hyper/Net, see Chapter 6).

Hyper/J automatically creates a *Class File* dimension from the set of units introduced into hyperspace using the project specification. Each class introduced into hyperspace has its respective concern in this dimension. Each respective hyperslice is populated by the appropriate class and its members. The *Class File* dimension allows the visualization of hyperspace from the classic perspective of OOP.

Concern mapping files are used to describe how the units that were previously introduced into hyperspace are mapped to concerns. This populates the corresponding hyperslices. Different types of units can be mapped to concerns. In fact, all kinds of compound and primitive units in Hyper/J hyperspaces can be mapped. Each mapping is declared by identifying a unit and its target concern (in a particular dimension). The order of declaration is important. A unit that has already been mapped to a concern can be declared in other mappings. If these declarations map the unit to concerns in the same dimension of previous declarations, the previous mappings are dropped and this new one is introduced. Otherwise, the unit is mapped to all other declared concerns, given they do not have any dimensions in common. This way, units can be mapped to more than one concern but cannot be mapped to different concerns in the same dimension. There can be mapping declarations for units that are contained in compound units that were already mapped themselves. In such cases, the units contained in compound units are only mapped to the concern of their specific declaration. This overrides the mapping imposed by their containing compound unit. It allows units contained in compound units to be mapped to concerns different from the one the compound unit is mapped to.

There is no explicit declaration for concerns and dimensions. Concerns are implicitly declared as the target of unit mappings. That is, if a new mapping references a non-existing concern, it is created. Furthermore, if the dimension of a new concern does not exist, it is also created. The model for MDSoC dimensions defined in the beginning of Section 3.2.1 determines that each dimension must contain all units. To achieve this, Hyper/J automatically creates None concerns in each dimension and populates them with the units that are not mapped to any concerns in the dimension. It is also possible to explicitly introduce units into None concerns.

In terms of the hyperslice implementation approach, as described in Subsection 3.2.1, Hyper/J supports all three hyperslice implementation approaches. Code can be physically decomposed by the developer before being introduced into hyperspace. It can also be used without any decomposition at the code level, by mapping units to the appropriate concerns. The hybrid approach is thus possible by mixing the two approaches. This is expected to be the most frequent approach. With it, all physically decomposable units can be decomposed in the source code. Yet, any units that are not physically decomposable can be mapped to more than

one concern in hyperspace. [Tarr01] proposes that physical decomposition should be done whenever possible.

Physical decomposition will bring severe advantages for developers, who will be able to manipulate decomposed code organized into concerns. Nevertheless, there may be some disadvantages of prioritizing physical decomposition. Units of code that is physically decomposed can still be used in different hyperspaces. If the dimensions of those hyperspaces are severely different from the dimensions used to decompose the original code, mapping might be difficult. Furthermore, hyperspace organization may change. A physical decomposition approach is harder to adapt to these changes than a virtual mapping approach. Nevertheless, these disadvantages are not significant when compared with the great advantage of having developers manipulating units as they are organized in hyperspace.

Hyper/J does not implement physical decomposition by itself, but it can rely on OOP decomposition to achieve this physical implementation. [Tarr01] (Subsection 4.3.1) proposes and promotes the use of a particular physical implementation for hyperslices: "Hyperslice Packages". In this implementation, each hyperslice will have a corresponding package in the code. As these packages (hyperslices) must be compiled prior to composition using Hyper/J, the Java compiler imposes a completeness constraint for each package. The units referenced from each hyperslice package must exist in the compiled code. Units outside of the hyperslice package can be referenced from other packages that offer them. [Tarr01] advises against this approach and proposes that each hyperslice package should instead be declaratively complete. The declarative completeness of each hyperslice package guarantees that it will compile without problems even if compiled without any other packages. Thus, declarative completeness achieves the requirements of the completeness constraint imposed by Java compilation.

[Tarr01] (Subsection 4.3.2) presents a process for achieving declarative completeness. It can be used in a physical decomposition model with hyperslice packages or in a virtual decomposition model, by matching the appropriate units to each concern/hyperslice. Declarative completeness in a hyperslice is achieved by introducing declarations for all units that are referenced and do not belong to the hyperslice. These declarations can be introduced as abstract units, which limits their use, or, in the case of operations, as operations that throw a predefined Hyper/J (unimplemented) exception. [Tarr01] does not present a way of introducing field declarations and Java does not offer abstract field declaration.

5.1.2 Hypermodules

Finally, Hyper/J uses a hypermodule specification file to define composition and achieve composed output [Tarr01] (Subsection 4.2.3). Each hypermodule specification file identifies the set of hyperslices used for composition, along with a composition strategy and other adequate relationships. Hyperslices are explicitly identified by name. Nevertheless, the identification of hyperslices in each hypermodule declaration file is optional. When this identification is not done, the hypermodule will contain all hyperslices in hyperspace, except for the hyperslices of None concerns and hyperslices in the *Class File* dimension.

Hyper/J offers three different composition strategies: *mergeByName*, *nonCorrespondingMerge* and *overrideByName*. Each composition strategy defines a different composition rule with its own combination of composition function and different way to match input units. Each composition strategy is detailed further on. Only one composition

strategy is allowed per hypermodule, so each hypermodule only has a composition rule. Hypermodules may include as many exception composition relationships as required.

Hyper/J composition strategies can match units in two different ways: *ByName* and *None*. With *None*, no units are matched and composition relies only on exception composition relationships. Each unit in the input hyperslices is simply output on its own, unless it is part of an exception composition relationship. With *ByName*, units are matched with units of the same unit type (classes with classes, interfaces with interfaces, and so on) that have the same name. If the direct container of a compound unit is a package, this compound unit is matched (by name and unit type) without regard to its containing package[12]. As for the remaining units, they are only matched if their containing compound units match or are the same. Yet, this matching function is not recursive. Composition functions are in charge of further matching units inside matched compound units.

The three composition strategies use two different composition functions: *Merge* and *Override*. *Merge* can be applied to sets of compound units of the same unit type. In this case it generates a single compound unit of that unit type containing the union of all units contained by the input compound units, except for any units which also match. These contained units that match are merged and the result is output to the generated compound unit. For example, two matching classes (*Fish1* and *Fish2*), each containing a *Breath* method, will be merged into a single class with a *Breath* method, that results from merging the two *Breath* methods. If *Fish1* has a Swim method and *Fish2* does not, the resulting class will have the *Swim* method from *Fish1*.

Merge can also be directly applied to primitive units. In the case of methods it generates a method that is equivalent to calling all of the matching input methods. The return value of each input method is also merged. By default the merged method returns the return value that would be obtained by invoking the last input method. It is also possible to define that a particular *Merge* composition function uses a summary function to compute the merged return value. In this case the return of the merged method is computed from each return of the input methods using the desired summary function. Hyper/J only allows declaring a summary function when *Merge* is used as the composition function of exception composition relationships and not as part of the composition strategy. These summary functions must be static and exist in the result of composition.

The order of the input units provided to composition functions is important. In the case of *Merge*, this order determines the order by which input methods are invoked in the resulting composed method. The order of input units is equivalent to the order of declaration of their containing hyperslices in the hypermodule. Between two units, the one that will come first is the one that belongs to the hyperslice which was declared first in the hypermodule. This order is only changed by the *Order* construct, introduced in hypermodule declarations as a specific relationship. The *Order* construct dictates explicitly that a particular unit precedes another, overriding any order obtained from the corresponding hyperslice declaration order.

The *Override* composition function for compound units is equivalent to *Merge*. It is also recursive and has its distinguishing result only when applied to primitive units (directly or by recursion from compound units). In such case, the *Override* composition function outputs only the last primitive unit from the tuple of input units. As such, ordering is even more critical for override composition.

[12] If this did not happen, no matching would be achieved. Java compilation already matches packages with the same name. Furthermore, it does not allow the declaration of units of the same unit type with the same name inside the same package.

Each composition strategy provided by Hyper/J implements a different composition rule with a different set of matching and composition functions:

- *mergeByName* uses the *Merge* composition function and the *ByName* matching function.
- *nonCorrespondingMerge* uses the *Merge* composition function and the *None* matching function. This is equivalent to defining no composition rule and that the default function used in exception composition relationships is *Merge*.
- *overrideByName* uses the *Override* composition function and the *ByName* matching function.

Each hypermodule only defines one composition rule but can define as many exception composition relationships as required. All types of units, except for packages, can be involved in exception composition relationships. In Hyper/J, entire hyperslice composition is only achieved using composition strategies. Exception composition relationships provide a smaller granularity composition mechanism at class level and below. The input units of each exception composition relationship must be units of the same unit type. Their order follows the same rule used with composition rules, that is, the order of hyperslices with exceptions introduced by the *Order* construct. Different kinds of exception composition relationships can be expressed using a set of Hyper/J relationship constructs:

- *Equate* is used to match units that are not matched by the composition strategy. The input units are explicitly identified. The composition function used is the one defined by the hypermodule composition strategy.
- *Match* is equivalent to *Equate* but allows units to be identified using pattern matching, instead of needing to be identified explicitly. This way, with *match*, the input units depend on the units present in the hypermodule's hyperslices that match the specified pattern. The remaining behaviour is the same as for *equate*.
- *Merge* imposes a *Merge* composition function to a set of explicitly identified input units.
- *Override* imposes an *Override* composition function to a set of explicitly identified input units.
- *NoMerge* imposes an equality composition function[13] to a set of explicitly identified input units. This way, *NoMerge* can be used to avoid applying the hypermodule composition function (*Override* or *Merge*) to a set of matching units. *NoMerge* has no effect in hypermodules that use a *nonCorrespondingMerge* strategy.
- *Bracket* introduces a new composition function, also called *Bracket*. *Bracket* changes a set of methods to additionally call a before method, before running their body, and call an after method, at the end of their execution. The input units are the methods that should be bracketed along with the before and after methods. The methods that should be bracketed are identified using pattern matching. Additionally, Hyper/J allows specifying that these methods are only bracketed when called from a specific context: a call-site. Only one call-site is allowed but it can be anything, from a hyperslice to another method. The *Bracket* composition function is limited to method unit types.

[13] The output unit of an equality composition function contains all of the units in the input tuple, and only these. An equality composition function can be recursively applied to a set of units time and again without ever changing them.

5.1.3 Usage and reuse

Due to its characteristics, Hyper/J allows heterogeneous usage scenarios, namely:

- Being used to decompose existing code (though the decomposed form cannot be output as code).
- Introduce new features, developed in a decomposed form, into existing indecomposed code.
- Use compiled Java programs to compose with custom code, or even composing amongst themselves.
- Creating different flavours of an application by removing specific concerns and introducing new ones (mix-and-match functionality).

Furthermore, one of the aims of MDSoC is to promote reuse. Hyper/J satisfies this requirement by allowing both hyperslice and hypermodule reuse. Hyperslices can be used in any hypermodule created for the hyperspace they populate. Hyperslices can also be reused in different hyperspaces. In this case it is best to have physically decomposed hyperslices, namely, using the hyperslice package approach. Otherwise, introducing the hyperslice into the new hyperspace will require identifying the physical units of the hyperslice again and matching only these to the hyperslice in the new hyperspace.

Hypermodule reuse is achieved by introducing the output of a hypermodule into a hyperspace. This must be done explicitly. Furthermore, as hypermodules output compiled class files, there is no way to physically decompose this for usage in hyperspace. Hypermodule output decomposition must be done virtually. Even if physically decomposing hypermodule output was possible it would not be adequate. To cope with changes to the hypermodules, their output units can be easily regenerated. Reintroducing them into the hyperspaces where they are used should not require any manual intervention, namely having to do or adapt a physical decomposition. Finally, it is discussable if any hypermodule output decomposition should be done at all. Hypermodules can be reused in hyperspaces and be involved in compositions at a higher level of abstraction than that of the input units of the hypermodule. In many of these cases each hypermodule will correspond to a specific concern and may be used just like a hyperslice, thus, no decomposition needs to be done.

5.1.4 Limitations

Hyper/J allows all the previously presented development scenarios and more. It also provides interesting reuse scenarios. Nevertheless, there are some limitations to how Hyper/J implements the MDSoC model:

- First of all, Hyper/J is limited to the code artefact in a single formalism (the Java language). MDSoC design and analysis implementations should be used together with Hyper/J to provide adequate traceability between artefacts.
- When using physical decomposition of code, the automatic *Class File* dimension created by Hyper/J will be useless. Recall that with physical decomposition, the problem domain classes are decomposed into several different classes according to different dimensions of decomposition. This is an effect already discussed in Section 2.4, where the dominant decomposition is compromised. In this case, the *Class File* dimension should be explicitly

declared (as any other dimension can be created using Hyper/J). This would recreate the lost object dimension.

- Finally, code must compile before Hyper/J is used. This might introduce limitations to the usage of Hyper/J, yet, it usually does not. In case virtual decomposition is used, code can be created in a standard fashion, thus compilation is not a problem. Otherwise, if physical decomposition is used, hyperslice declarative completeness usually satisfies the compiler requirements.

[Tarr01] (Section 4.5) presents other, technical, limitations of Hyper/J. We summarize the effects of these technical limitations, especially in regards to the MDSoC model:

- The *nonCorrespondingMerge* strategy does not work. This means that hypermodules will always compose units, matched by name, using either an *Override* or *Merge* composition function. There may be need for a hypermodule that only composes units with exception composition relationships. Without *nonCorrespondingMerge*, to achieve such a hypermodule, all unwanted matching units should be overridden with a *NoMerge* construct.

- *Merge* and *Override* constructs are not available for exception composition relationships. This way, exception composition relationships are limited to using the default composition function defined by the composition strategy or the *Bracket* composition function.

- Other constructs (like *NoMerge* and *Order*) are limited to specific types of units. This limits the expressiveness of composition.

- Pattern matching also has some limitations. For instance, the pattern matching used in mapping units to concerns does not allow recursion when applied to packages. Most of the limitations with pattern matching are easily overcome, by writing more detailed relationships.

- All output units, except for fields, are declared public, independently of their visibility modifier in their origin hyperslices. This is fruit of an additional transformation of composition with Hyper/J. This transformation is unwanted in regards to the MDSoC model and should be removed from Hyper/J.

- All output units will belong to the same package. This is irrelevant if the units in hyperspace are physically decomposed using hyperslice packages. Yet, in complex systems it might be useful to output several different packages. This would for example, facilitate hypermodule reuse. The MDSoC implementation that is presented next (HyperC#) has a similar limitation, but at a much more serious level: it can only output a single class unit.

[Tarr01] does not identify any origins for these limitations other than technical ones. This way, these issues should be overcome by corrections to the Hyper/J implementation. Then, the full power of a versatile and reusable MDSoC implementation for Java would be unleashed.

5.2 HyperC#

HyperC# [Hantelmann06] is an MDSoC implementation for the C# language. Even though HyperC# is implemented as a pre-compilation process, it relies on a graphical user interface (GUI) for the manipulation of code and the definition of composition. This GUI must be used to create C# classes that can be composed. The same GUI supports the definition of an MDSoC hyperspace with dimensions and concerns. Finally, the GUI is also used to define hypermodules and obtain their output.

In the class definition stage, the only purpose of this GUI is to gather meta-data about the code, avoiding the need to use a code parser. The gathered meta-data is stored in an XML file that should always accompany the original class file while HyperC# is used. For hyperspace definition, the GUI allows the declaration of dimensions and their respective concerns in a visual form. Then, the GUI allows the loading of classes that have the respective XML meta-data file. Methods and constructors in these classes can then be placed inside the appropriate concerns of each dimension. This way, HyperC# allows the decomposition of class units while populating the hyperspace defined in the step before. The definition of hyperspace achieved this way can also be saved to a XML file. Next, the same GUI is used to define composition, in the form of hypermodules. First, a default composition action is defined. This determines the composition rule used for the hypermodule. Additionally, it is possible to define, at most, two exception composition relationships for the methods being composed. Finally, the GUI allows the generation of composed output code, which is limited to a single class, and its compilation.

While defining the hypermodule, alternatively to defining a composition action, the GUI allows choosing that no composition will be done. In this case, the output is the decomposed code, according to the concern mapping done in the prior stage. This can be used to obtain physically decomposed code for existing classes.

HyperC# is an MDSoC approach that is limited to the code artefact and a single formalism: the C# language. It offers a set of composition constructs for methods, but only fixed class composition. This way, methods can be created and introduced into hyperspace in a decomposed form. Latter, they can be composed into a single method again, although there are several other composition alternatives. Due to this liberty, methods can be considered primitive units in HyperC#.

5.2.1 Dimensions, concerns and hyperslices

Like Hyper/J, HyperC# does not offer a physical implementation of hyperspace for programmers to develop in. The GUI allows the matching of developed methods to concerns. Unlike Hyper/J, HyperC# only allows the matching of method units to concerns. The mechanism used to achieve this decomposition seems to implement a virtual decomposition approach (the second approach in Subsection 3.2.1). Nevertheless it implements this kind of decomposition on top of a physical implementation. This limits the virtual decomposition approach and removes some of its advantages. Namely, the same method cannot be mapped to more than one concern, even in different dimensions. This forces methods that overlap different concerns to be physically decomposed prior to being introduced into hyperspace. Contrary to HyperC#, and according to the MDSoC model, Hyper/J allows matching the same unit to different concerns in different dimensions. This is a powerful aspect of the virtual decomposition approach that is missing in HyperC#. It is this way that Hyper/J provides support for indecomposable units belonging to more than one concern.

In HyperC#, the physical hyperslice implementation code can be obtained by skipping composition, after all necessary class methods are matched to concerns. In the decomposed code, for each class that was used in decomposition (origin class) there will be a corresponding class (concern class) in each concern that has methods from the origin class. At most, there are as many decomposed classes as origin classes times the number of concerns. If the original inheritance information is retained in this decomposed form, HyperC# is able to retain the physical object dimension in its hyperspaces. Nevertheless, [Hantelmann06] does not provide information on whether inheritance is kept or lost in hyperslices.

With its decomposition model and the feature for physical output of decomposed code, HyperC# decomposition can be used in three different ways:

- Classes and their units can be created decomposed according to the hyperspace that will be created with HyperC#. When the hyperspace is defined, they can be introduced into it in this decomposed form. This way, the code will be manipulated in its decomposed form, with the inherent advantages.

- Non-decomposed code can be loaded, after the hyperspace dimensions and concerns are defined, using HyperC# to decompose it. This provides aid in decomposing existing code. The existing code can still be manipulated in that form and used in different hyperspace definitions. This has the disadvantage of always needing to decompose the code while defining hyperspaces. Yet it can be differently decomposed for each hyperspace.

- Finally, the result of decomposing existing code with HyperC# can be materialized into decomposed code using the appropriate HyperC# output feature. This code can then be manipulated in its decomposed form like in the first approach.

HyperC# also introduces in each concern class the global declarations of the origin class. This partially addresses the declarative completeness requirements of hyperslices (concern classes). [Hantelmann06] is not clear regarding which global declarations are involved. Nevertheless, these will not include method declarations, as [Hantelmann06] proposes a different approach for declaring referenced methods. Methods that are referenced from concerns in a dimension, and do not belong to the dimension, should be included in the None concern of that dimension. This will not be possible with HyperC# due to the limitation that a method can only be put inside one concern. The remaining referenced methods can be introduced into None concerns without problems, as they are not present elsewhere in hyperspace. Nevertheless, this approach for methods does not provide declarative completeness of hyperslices but of the result of composition, that is, of the hypermodule output set.

To appropriately achieve hyperslice declarative completeness with methods, the first HyperC# decomposition usage scenario from the three presented above should be used. Declarations for referenced methods should be introduced in the appropriate physically decomposed classes. That is, the classes that correspond to the concerns that reference the method. When class methods are matched to the respective concerns all of the referenced methods will be present in that concern. Finally, composition should use the override construct to impose the implementation of the method over its declarations. The limited composition model of HyperC# will not support declaratively complete hyperslices in most hypermodules, except for ones that use an *OverrideByName* composition action. This happens because HyperC# only allows using override composition if the *OverrideByName* composition action is chosen for the hypermodule.

5.2.2 Hypermodules

Hypermodules can be created for each hyperspace defined with HyperC#. In HyperC# hypermodules, the set of hyperslices is always equal to the set of hyperslices in hyperspace. Nevertheless, it is possible to create a hypermodule with a subset of the hyperslices of a particular hyperspace. The original hyperspace should be edited, removing unwanted hyperslices and saved as a new temporary hyperspace. This hyperspace should then be used to create a hypermodule which will only have the appropriate hyperslices.

Like Hyper/J, HyperC# offers three different composition strategies: *MergeByName*, *NonCorrespondingMerge* and *OverrideByName*. As for composition functions, HyperC# offers the same ones as Hyper/J: *merge*, *override* and *bracket*. HyperC# supports a single composition strategy for each hypermodule. It also supports exception composition relationships. The HyperC# GUI allows the creation of at most two exception composition relationships per hypermodule. One can use a *bracket* composition function, defining as input units the method to be bracketed along with before and after methods. The other exception composition relationship is called *equate*. *Equate* simply uses the composition function defined by the composition strategy (either *override* or *merge*). It defines as input units explicitly identified methods (using the GUI), which must have the same signature. Both exception composition relationships are optional.

HyperC# allows the creation of different hypermodules for the same hyperspace definition. This way, hyperslices can be used in different compositions, providing hyperslice reuse. Hypermodule reuse is also possible, by introducing the output of a hypermodule in a new hyperspace. The only limitation to hypermodule reuse is that their output will be a single class (in source code form). This will require creating a different hypermodule for each desired output class and reusing these classes one by one.

5.2.3 Limitations

HyperC# may be an interesting MDSoC implementation for .NET but it has serious limitations that may affect its usability in real-world applications:

- HyperC# needs to use a specific GUI to offer MDSoC support. This might be acceptable for hypermodule declarations, but not for creating classes. If HyperC# used a parser on existing source code files, it could generate the same XML metadata it gathers from the class creation GUI, allowing the use of existing software for creating code.

- The object dimension is lost with HyperC# composition. Using a different hypermodule for each desired output class overcomes this limitation. Still, this work-around will be impracticable in applications with more than a few output classes.

- More generally, this approach is limited to manipulating (decomposing and composing) methods. There should be decomposition and composition constructs for other units, namely classes, interfaces, variables and properties.

- Aside from the *bracket* composition function, it is only possible to define exception composition relationships with the composition function defined in the composition rule (using the *equate* construct). It should be possible to define *override* and *merge* exception composition relationships independently of the composition rule being used.

- As for exception composition relationships, each hypermodule is limited to one *equate* and one *bracket*. Any number of exception composition relationships should be allowed.

- Finally, the merge composition implementation creates a new method concatenating the bodies of each matched method. The resulting method shares the same scope for all the concatenated method bodies. Because of this, programmers must be aware of the composition used with their methods and avoid using the same variable names, etc. This is usually impracticable. As stated previously, this limitation should be overcome by introducing separate scopes for each merged method body.

5.3 Conclusions

Both Hyper/J and HyperC# are limited to applying MDSoC to code in a single language (Java or C#). Hyper/J works with compiled Java class files, while HyperC# works with source code. Both promote the reusability of hyperspace elements. Defining composition with both approaches has a similar basis: using a composition strategy per hypermodule along with exception composition relationships. Yet, the possibilities are much more limited in HyperC#, which only supports two exception composition relationships and uses fixed class matching to output a single class. Finally, HyperC# forces developers to work in a specific GUI that is very limited while Hyper/J allows developers to remain using their development environment.

Chapter 6

Hyper/Net: An MDSoC solution for .NET languages

Hyper/Net is a pre-compilation compositor or weaver for an MDSoC hyperspace created with .NET code. This means Hyper/Net processes .NET code that is in a decomposed form. It uses it to generate code that can be compiled using a normal .NET compiler. Hyper/Net was developed as part of the material support of the thesis presented herein.

Hyper/Net is based on the fact that .NET code decomposition can be achieved using a native feature of .NET languages: partial types. Decomposition using partial types alone is enough to implement a basic MDSoC hyperspace. Partial types are able to offer decomposition and composition facilities at the granularity of classes.

This chapter provides a presentation of the Hyper/Net MDSoC approach, bridging it to the MDSoC model presented in Chapter 3. The first section presents how we used .NET partial types to create a basic MDSoC hyperspace model. Advantages and limitations of this model are analysed. The MDSoC model is used to show how the hyperspace created with partial types is in fact an MDSoC hyperspace. The second section presents how Hyper/Net's composition functionalities are used to extend this model, which limitations are overcome and which remain.

This chapter is closely related to the two that follow it. Chapter 7 will present how Hyper/Net can be used by programmers, how it is integrated with IDEs and also two implementation examples using Hyper/Net. Finally, Chapter 8 will present the details about the Hyper/Net MDSoC composition process and the Hyper/Net implementation.

6.1 The Partial Types MDSoC approach

Sections 5.1 and 5.2 present two MDSoC implementations. Both are extensions to an OOP language and introduce constructs for both the decomposition and composition stages of MDSoC. This section shows how .NET partial types can be used as an MDSoC implementation. As partial types are a native feature of .NET 2.0 languages, this is the first native MDSoC implementation, at least to be acknowledged as such.

.NET languages are mostly object oriented, though there is a tendency to some becoming heavily multi-paradigm[14]. The following model uses partial types as a secondary decomposition mechanism that complements the dominant object decomposition in these languages. The additional decomposition with partial types will allow the creation of a multi-dimensional concern space.

Partial types allow the separation of a type (class or interface) between several different files, with a partial type (class/interface[15]) for each file. If there is a directory for each concern at hand, then, types can be separated according to these concerns. The respective partial type files can be placed under the directory for the appropriate concern. This way, programmers will be able to work separately in each concern by working on files in the respective directory. Nevertheless, concerns do not exist isolated, they will belong to dimensions. Dimensions can just as well be implemented as directories. Each concern belongs to one dimension. The directory for a concern will exist inside the directory of the dimension it belongs to. This way, with a two level directory structure, populated by partial types at the second level (the concern level), it is possible to create an MDSoC hyperspace in all .NET 2.0 languages. In this hyperspace, .NET compilation is in charge of composition by merging the partial classes of each class type into a single class. Figure 6 provides an example of such an MDSoC hyperspace directory structure with two dimensions, holding two concerns each, and partials of the *Class1* class existing in three of its concerns.

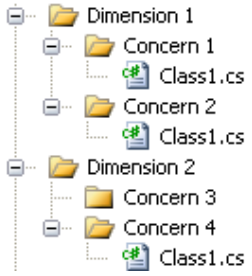

Figure 6. Example of a directory structure implementing a 2D hyperspace with a single class.

6.1.1 Dimensions, Concerns and Hyperslices

The MDSoC hyperspace implemented using the approach summarized above is limited to the code artefact and offers a choice of different formalisms: all .NET 2.0 (and above) languages. Unless .NET should offer language coexistence mechanisms for the same project, only one of these formalisms is supported in each MDSoC hyperspace at a time.

Partial types allow classes to be decomposed. Class decomposition results in partial classes. Partial classes abide to a lighter set of completeness constraints than classes. Thus, partial class units can be considered at a smaller granularity than class units. A partial interface can similarly contain a sub-set of the declarations of the complete interface. Yet, there are no incomplete interfaces, so partial interface units have the same granularity of interface units. However, partial types allow interface declarations to span several different files, also providing a decomposition mechanism for interfaces.

[14] For example, C# 3.0 is heavily extended with functional programming constructs.
[15] When possible we refer generally to partial types instead of specifically to partial classes or interfaces. When adequate, details about partial classes or interfaces are presented.

Method units are at an even smaller granularity. With partial types, it is possible to have a partial class with only one method. This method can also exist in different partial classes for the same class type. This way, partial types also provide a decomposition mechanism for methods[16]. But there is no native method composition mechanism in .NET. Methods decomposed into different partial classes will not be composed and make it impossible for these partial classes to be composed themselves. In these situations, the .NET compilation process yields an error due to finding more than one method implementation for the same class type. In the model implemented by the partial type approach, method decomposition is possible but invalidated by the compilation process. As such, the primitive units of this model are partial types (classes and interfaces).

As seen in Section 3.2, the first step in applying MDSoC is defining the hyperspace. First, dimensions and their concerns are determined (step 1 in Figure 7). The two-level directory structure for dimensions that we proposed easily accommodates the requirements of MDSoC dimensions and concerns. Implementing this directory structure is step 2 in Figure 7. For instance, the MDSoC model imposes that all dimensions and concerns must be unique. Furthermore, concerns can only exist in one dimension. This is guaranteed by the properties of the two-level directory system used to implement dimensions and concerns in the current approach.

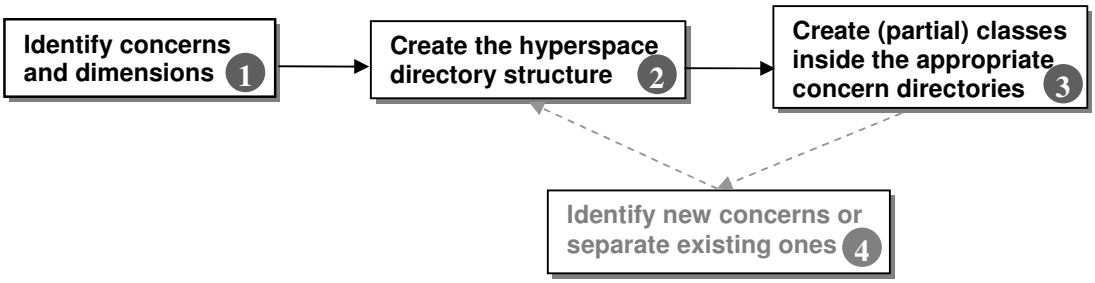

Figure 7. Simplified block diagram for the .NET partial types MDSoC approach.

The primitive units of this approach (partial types) can be physically separated in different files, each placed under the appropriate concern directory (step 3 in Figure 7). This way, to populate the concerns, existing classes and interfaces should be decomposed into partial types and new units should be created as partial types, unless the entire type would belong to a single concern.

In this approach, concern directories are hyperslices, as they physically implement the concern boundaries. There is an equivalence correspondence between concerns and hyperslices, as [Tarr99] proposes and Hyper/J implements [Tarr01]. This equivalence is the most limited implementation of the more general concern-hyperslice relationship of the MDSoC model.

An MDSoC hyperspace can be extended by the introduction of new units, repeating step 3 in Figure 7 for each new unit. It can also be extended by the creation of new concerns and even dimensions, through the optional step 4 in Figure 7. These extension features are trivial in the partial types approach. New units can be introduced by being placed in new files inside the

[16] In fact, simply by splitting up methods, even outside of partial classes, we are decomposing them. OOP provides most of these decomposition functionalities. What it does not provide are the respective composition mechanisms. Without composition mechanisms, the result of decomposition is invalid for compilation.

appropriate concern directory. If the units that are being introduced correspond to more than one concern they should be decomposed using partial types. Dimensions or concerns are added by creating new directories in the appropriate level. These are placed under the parent dimension directory in the case of concerns.

This approach only offers one kind of hyperslice implementation model. That is, the first implementation model presented in Subsection 3.2.1, which implements actual physical decomposition. .NET offers no kind of mapping facilities to create virtual hyperslices while retaining composed code. After the source code is decomposed, or as it is created in the decomposed form, there is no way to manipulate it in a composed form. As the composition mechanism is part of the compilation process it will not even be possible to view the composed code directly. This way, most of the time, the source code must be manipulated while decomposed. Programmers using this approach will be forced to focus each concern separately. In our opinion this is mostly a good thing.

Nevertheless, when introducing new types or when composition itself is under scrutiny, there should be some kind of view of the composed result. Microsoft Visual Studio offers two features that will help in this matter; both were presented in Section 4.2.5. In Visual Studio 2005, class diagrams ignore the partial class structure and display entire classes, thus class diagrams offer a perfect view of the composed result. This view is equivalent to the object dimension view. As such, class diagrams in this approach can be considered as a materialization of the object dimension of the MDSoC model. Class diagrams also allow navigation to the code containing the units displayed for a particular class (variables, methods and properties). In the case of types decomposed in partial types, this navigation is made directly to the partial type containing the unit in question. As for the second feature, Intellisense is an auto-completion feature that suggests units (methods, classes, etc.) based on the current context of program edition. Intellisense uses information that is equivalent to the compiled result of the project. When programming in a .NET project that is organized as a hyperspace, this information is equivalent to the result of composition. As such, Intellisense also provides a contextual view of the object dimension. A feature equivalent to Intellisense is also available in the SharpDevelop IDE with the same behaviour for partial type based hyperspaces.

Without mapping facilities and relying solely on physical decomposition, the current approach only supports the separation in different concerns of units that are decomposable. But, there can be indecomposable units that may need to be associated with different concerns in different dimensions. These units must be left in only one of these concerns or otherwise be replicated in the different concerns. Both options are unsatisfactory. This approach is limited to the code artefact, so different artefacts, that usually must be bound using a virtual scheme, are not contemplated.

As seen, with this approach, units that belong to more than one concern are either decomposed into these different concerns or are indecomposable, and can only be bound to a single concern. In this approach, the same unit never belongs to two concerns at once, thus, concerns never overlap. This provides the necessary guarantee that no overlapping occurs between concerns in the same dimension, fulfilling one of the requirements of the MDSoC formal model [Ossher99]. Nevertheless, as justified in the previous paragraph, this also makes it impossible for concerns to overlap between dimensions, even though it is allowed in the MDSoC model [Ossher99].

Recall from the MDSoC model, that all units must belong to each dimension. As such, in MDSoC, each dimension has a special None concern [Ossher99]. In this approach each unit will exist in only one dimension. This causes the None concern of each dimension to be made

up of all the units that belong to all the other dimensions. That is, the None concern of a dimension will be composed by all the files in all the directories of hyperspace, except the directory for the dimension in question. This way, instead of manually creating a None concern directory in each dimension, with this approach, the remaining dimension directories should be used to find None concerns for any dimension.

MDSoC also defines that all units must belong to at least one hyperslice [Ossher99]. In this approach, units will belong to a hyperslice as long as they are placed under the appropriate directory. It could be dictated that no files can exist outside concern directories, but this is not enforced natively. Hyper/J does not enforce this restriction either [Tarr01] as it would forbid iterative refactoring from composed code into MDSoC decomposed code.

MDSoC defines that hyperslices must be declaratively complete [Ossher99]. This is usually achieved by introducing declarations for all referenced units that are not present in the hyperslice. Here, the partial type approach is limited by the fact that the .NET compiler does not allow multiple declarations of units inside the same project. Most .NET languages also do not allow the declaration of units without also providing an implementation, unless these units are declared abstract[17], which is not adequate for this approach. Instead, we introduce declarations as empty or equivalent implementations. For instance, a method declaration can be a method that throws an unimplemented exception. This way, with partial types, class and interface units can always be declared as an empty partial type, overcoming this limitation for classes and interfaces. If the referenced units do not exist inside the project, then, it is possible to introduce declarations for any kind of units (for example, classes, methods, variables and properties). This way, in this approach, declarative completeness in hyperslices can only be achieved when referenced units do not exist inside the project, are classes or are interfaces. Declarative completeness in hyperslices cannot be achieved when there are references to class members that exist elsewhere inside the project. Hyper/Net overcomes this limitation for method units as we will present in the next section. Nevertheless, without declarative completeness, when referenced units are missing, the compiler will show the situation as an error. It can be corrected by introducing hyperslices which offer the missing referenced units in the project or a reference to an external project with these units. The need for missing units is easily acknowledged this way. Having them explicitly declared might not provide a great advantage. Again, similarly to the issue with the physical None concern implementation, we are against the manual introduction of elements that can be automatically identified.

6.1.2 Hypermodules

Finally, having addressed most decomposition and hyperspace structure issues, we focus on the composition model of the partial types MDSoC approach. Composition is automatically provided in this approach. It is executed by the .NET compiler. During compilation, partial types are brought together into a single piece which holds the entire type implementation in the compiled code. Partial types implement strict unit matching by type. A partial class matches other partial classes of the same class type. The equivalent applies to interfaces. There is no way to compose partial types that do not correspond to the same type. The integration process is simple[18]: the units (methods, variables, properties, etc.) from each corresponding partial type are brought together under a single new type. This is a kind of additive integration and shares the attributes of merge integration from SOP and MDSoC.

[17] For example, abstract methods can only exist in abstract classes. Abstract classes cannot be instantiated, thus the declared method cannot be referenced.
[18] It was already described in full detail in Subsection 4.2.3.

In this approach, composition is defined by the decomposition, that is, by the partial types that are created during decomposition. The composition model for partial types is equivalent to defining a single MDSoC hypermodule for each .NET project. This hypermodule will be composed of the set of all hyperslices (concern directories) in the project that is being compiled. The compilation of a .NET project provides composition relationships that bind the different partial types of the same type to a single unified type. From this perspective, partial type composition can be seen as a composition/weaving process taken during .NET compilation. .NET compilation adds a limitation to the hypermodule defined by a .NET project. Unless the resulting hypermodule is declaratively complete, compilation will yield missing reference errors and fail. For the resulting hypermodule (project) to be declaratively complete it will have to reference only units that it contains, are contained by the referenced projects and binaries or are somehow declared. Hyperslices can reference any units offered by other hyperslices inside the project, this will not cause any compilation errors. Thus, hyperslices do not need to be declaratively complete themselves.

The hypermodule for each .NET project will contain all of the hyperslices in the project, that is all of the concern directories included in the project. Then, there will exist a composition relationship for each defined type that is decomposed in more than one partial type declaration. Here, the input units are all partial types of the type being composed; in any order[19]. The composition function will generate a single output type. The composition function implements a simple merge composition for classes and interfaces. The output class or interface will contain the union of all unit sets of each partial type, that is, all of the methods, variables, properties, etc. of all the partial types being composed together. This composition function has the requirement that there are no duplicate unit signatures in all of the input units (in partial types).

6.1.3 Model Limitations

Recall, from the declarative completeness considerations for this approach, at the end of Subsection 6.1.1, that it is possible to introduce declarations for referenced units that do not belong to a project or to referenced projects. If these kinds of declarations are introduced, the .NET compiler will output valid code that is also a valid hypermodule. Nevertheless, during runtime, when units that were only declared are used, errors will occur. For instance, a method declaration can simply be a method that throws an unimplemented exception. At runtime, using this method will yield an unexpected exception. Thus, these resulting hypermodules are incomplete. To create complete hypermodules out of these incomplete ones we would require a mechanism capable of composing units in different compiled projects. It is important to note that compilers for the .NET 2.0 languages do not allow partial types to span different projects. That is, all partial classes of the same class type must belong to the same project. This happens because .NET uses referenced projects in their composed form. If it was possible for partial types to span different projects, it would introduce added flexibility, namely the ability of composing units in different projects. Due to this .NET limitation, hypermodules are not reusable in new compositions, thus must be complete (contain all referenced units) to be of any use. The MDSoC model offers hypermodule composition to promote reuse, but, as we will see, the partial types approach only offers other reuse mechanisms, namely at hyperslice level.

[19] The order of the partial types is not important as the composition function implemented by .NET partial types is commutative.

As presented, the matching and integration executed by this approach is defined by partial type decompositions. Additionally, our composition engine is the compiler, which generates output for a single entire project at a time. This way, a particular project is limited to defining one hypermodule. In MDSoC it is possible to use the same units in different hypermodules. Thus, units inside a particular project should be usable as code in other projects. The MDSoC model supports multiple distinct hypermodules for the same hyperspace [Ossher99]. Notice that for this approach, up to this point, we have left the set of all units in hyperspaces undefined. We will now bind it, making it possible for several hypermodules to co-exist in the same hyperspace, thus, gaining added hyperslice reuse flexibility for the approach.

We could define that the unit set of each hyperspace is composed of the units inside a particular project united with all the units in referenced projects. In this case we would be limiting our hyperspace to a single project and thus a single hypermodule. Instead, we can broaden the set of units in our hyperspaces. Because hypermodules are implemented as projects, to follow the MDSoC model, we need to include as many projects as required into our hyperspace. This is simply done by broadening the set of units belonging to hyperspace to include any given amount of different projects. To do this, the dimension/concern space model that was initially presented has to be extended. In a hyperspace using this approach, containing more than one project, the set of dimensions is the union of all dimensions in all the projects. The same applies to the concerns inside these dimensions. Each dimension of such a hyperspace contains the set of all concerns belonging to that dimension in all of the projects in the hyperspace. The set of units in the resulting hyperspace is the union of all units in each project.

At this point, we are faced with a problem. Hyperslices should be usable in any of the hyperspace hypermodules; they are in MDSoC [Ossher99]. This means that directories from each project should be usable by other projects. The simple directory system in this approach does not allow this, as the directory structure for each project is separate from the other projects. Obviously, replicating hyperslice directories that are shared among projects is not an acceptable solution. If the file-system that supports the projects allows symbolic links then our problem is solved. One of the projects including a particular hyperslice can contain an actual directory with files while the remaining projects have a symbolic link to this directory (a linked hyperslice). Not all file-systems support such symbolic links, but even in these cases, there is the possibility of using a source control solution to implement such symbolic links. The directories could be replicated in the local file-system but changes in any of the corresponding directories would map to the same directory and respective files in the source control tree. The source control solution has the added advantage of working independently of the file-system implementation being used. With either solution, it is even possible to have a project fully composed of linked hyperslices. In the MDSoC model, hypermodules cannot define specific dimensions to which concerns belong, they use the dimensions defined in hyperspace. To implement the MDSoC model correctly, linked hyperslices should always be placed under the same dimension directory as the respective real hyperslice.

As for composition, the limited matching model that was already presented poses an additional limitation to this approach. Even though the previous paragraph describes how hyperslices can be placed inside any given hypermodule, the possible compositions with these hyperslices are severely limited. Units in hyperslices introduced in a particular project can only be composed with equivalent units that share the same type and are all partial types. When introducing hyperslices from different external origins, developed separately, it would be an enormous coincidence if the partial types of units to be composed matched this way.

With the extensions presented, each hyperspace will be composed by all of the units in a set of projects (and the respective units from all referenced projects). Furthermore, each project

will correspond to a different hypermodule. That means, there can be any amount of hypermodules in a hyperspace. Thanks to symbolic links, either in the file-system or source control, these hypermodules can be made up of any set of units from all the units belonging to the hyperspace. The MDSoC model also defines a hyperspace composed of a set of hypermodules to which any hyperslices in hyperspace can belong. With the help from symbolic links, the partial types MDSoC approach fully supports these aspects of the MDSoC model.

The most significant limitations of this approach relative to the MDSoC model can be summarized as follows:

- In this approach, concerns cannot overlap between dimensions like they can in MDSoC. This problem would be minimized if units smaller than classes were decomposable, but they are not.

- MDSoC offers reuse mechanisms at hypermodule and hyperslice levels. The partial type approach only offers a hyperslice reuse mechanism (with composition limitations) and does not allow hypermodule reuse in new compositions like MDSoC does.

- Finally, this approach is limited to a single artefact and one formalism at a time (for each hyperspace).

The next section presents a solution to some of these limitations.

6.2 The Hyper/Net MDSoC approach

The major advantage of the Partial Types MDSoC approach is that it is natively supported by .NET 2.0. Without additional programs, .NET developers can start using MDSoC in their software by using this approach. Nevertheless, this approach has limitations in terms of reuse and the granularity of decomposition. Hyper/Net is capable of addressing some of these limitations. This way, Hyper/Net is introduced as a complement to the composition features of .NET compilation, in particular, as a pre-compilation code processor.

In the partial types approach, partial types are used as a decomposition/composition construct for classes and interfaces. Even though methods are decomposable, the resulting code is invalid as there are no composition mechanisms for the decomposed methods. Hyper/Net introduces such method composition mechanisms with constructs that extend the partial types MDSoC approach. As we will see, this lowers the granularity of primitive units of the implemented model from partial types to methods.

The composition constructs of Hyper/Net take the form of .NET attributes. Attributes can be applied to any unit of code (class, interface, variable, etc.) but Hyper/Net only takes into account composition attributes for methods. The attributes made available with Hyper/Net are a kind of composition metadata for methods. The use of partial types introduces similar metadata for classes and interfaces. But, partial types also enable class and interface decomposition whereas method decomposition is achieved using language features, by creating separate, usually smaller, yet complete methods. Partial classes do not need to be complete classes.

With the adequate Hyper/Net composition attributes, there can be a set of methods with the same signature in different partial types of the same type. These methods will be composed

according to the relationship defined by the attributes. Ultimately, the attributes define the composition function to be used while the types of partial types and method signatures define which units should match. Together, partial types and Hyper/Net method attributes define composition relationships which will be detailed further on.

The aim of Hyper/Net is to compose methods. As seen in the previous paragraph, Hyper/Net will use attribute information from matching methods to execute their composition. Partial types do not allow methods with the same signature in partial types of the same type. This way, the code that should be processed by Hyper/Net cannot be handled by the .NET compiler. As such, Hyper/Net is forced to work as a code pre-processor that generates valid .NET code. Hyper/Net takes as input a .NET project and generates an equivalent body of code with all matching methods composed into code that follows the .NET compiler rules: thus is valid. To generate valid .NET code and compose methods, Hyper/Net has to replace each set of matching methods with only one method. To achieve valid .NET code, the composed method could be placed under any of the involved partial types, where the original methods were contained. This would also allow the use of the generated code to provide hypermodule reuse. This way, the code output of MDSoC might be used by developers in new compositions. Still, recall that in the partial type MDSoC approach, different partial types (and the methods therein) may belong to different concerns. Composed methods overlap all the concerns where the original methods belonged. Unless all matching methods belong to the same concern, there is no partial class where the composed method can be placed without violating the dimensions of the MDSoC model. To avoid this situation, Hyper/Net also implements partial type composition. Thus, Hyper/Net composes partial types, according to the .NET partial type model, into a single unified type. Matching methods in each composed partial type are composed and placed in the class resulting from partial type composition. Chapter 8 presents the details of this Hyper/Net composition process and its implementation. Finally, by implementing .NET partial type composition, Hyper/Net supports partial types in .NET framework versions prior to 2.0. This way, Hyper/Net incidentally offers partial type support for .NET 1.0 and 1.1.

In terms of expressivity, .NET partial types only enable one kind of composition, that is, merge composition for partial types, as described in the previous section. Hyper/Net composition attributes, which are used with methods, provide a wider choice of three different composition types. Two of them, override [Ossher96] and merge [Harisson96], existed in SOP and are supported in Hyper/J [Tarr01], whereas the other, bracketing, was introduced in Hyper/J [Tarr01]. In Hyper/Net, when the override composition attribute is applied to a method of a partial class, it will be the only method with that signature in the class resulting from Hyper/Net composition. The merge composition attribute, when applied to matching methods, dictates that each method is retained, with a changed name. The original methods are replaced by a single method with the original name, which invokes all of the matching methods, using a total ordering. This total ordering is defined by a different priority level present as an argument of the merge composition attribute of each method. Additionally, the merge attribute also dictates how the resulting method will compose the results from each matching method, using another argument. Bracketing is an exception, as it can only be applied to a single method. When the bracketing composition attribute is used, the method is only changed by additionally starting and ending with the invocation of two distinct methods that the attribute has to define. Each of these three attributes defines a different composition function; these are presented further on as we bind the Hyper/Net MDSoC approach to the model presented in Section 3.2.

6.2.1 Dimensions, Concerns and Hyperslices

The Hyper/Net MDSoC approach is still limited to the OOP artefact. Furthermore, Hyper/Net processes source code using a parser. The current Hyper/Net implementation uses a parser that is limited to two .NET languages: C# and VB.NET, but supports them starting from the 1.0 .NET framework. This way, the Hyper/Net MDSoC approach is limited to these two formalisms: C# and VB.NET.

With Hyper/Net, type decomposition is still achieved using partial types. Nevertheless, as Hyper/Net implements method composition, method decomposition is now supported. Methods can be decomposed into smaller methods, with the same signature. Even with Hyper/Net, it is not possible to decompose units into units smaller than methods. This way, methods are the smallest decomposable unit and at the same time one of the primitive units in hyperspaces using the Hyper/Net approach. As stated in Section 5.1, for Hyper/J, this is an exception to the definition of primitive units from Section 3.2.

The current Hyper/Net implementation only supports the composition of methods (other than constructors). Composition support could be extended to other types of units below the class level, namely constructors, variables and properties, using the same attribute based approach. This is one of the issues focused in Section 10.3.

Hyper/Net introduces only a physical decomposition mechanism for methods. It offers no mapping mechanism to place the same method in different concerns/hyperslices of hyperspace. This is equivalent to what happens with the partial types approach, for classes and interfaces. Recall that in the partial types approach, hyperslices are composed of types and partial types inside files in the corresponding concern directory. With Hyper/Net, a method unit can be placed inside a hyperslice by adding it to a partial class in the appropriate directory. If the directory already has a partial class for the class type to which the method belongs, this partial class should be used. Otherwise, a new partial class (of the class type that the method belongs to) should be created and the method placed inside it. The remaining units are similarly matched to a hyperslice, like they were with the partial types approach. The only difference with Hyper/Net is that methods with the same signature can exist in different partial classes for the same class type.

6.2.2 Hypermodules

With the Hyper/Net approach, hypermodules are still defined as .NET projects. The .NET project composition rule, defined by the partial types approach, must be extended to support Hyper/Net method composition. Hyper/Net method composition provides additional integration by offering a new composition function for each type of composition offered: merge, override and bracket. These composition functions are exclusively for use with method units. The composition rule itself is extended by adding one exception composition relationship for each set of matching methods. The attributes of the methods in the set define the composition function to be used in the exception composition relationship.

There is a set of matching methods for each set of methods, belonging to partial types of the same type, where the methods have the same signature and at least one of the methods has a Hyper/Net composition attribute. In this case the composition attributes must be either merge or override. Bracket composition attributes define a different set of matching methods, with a bracket attribute and the two methods declared as before and after methods in the attribute. Additionally, the following rules must also be followed:

- In case one method in a set of matching methods has an override attribute, no other method in the set can have an override or merge Hyper/Net composition attribute.

- In case one method in a set of matching methods has a merge attribute, all other methods in the set must also have a merge attribute with a different priority level (Section 7.2 provides details about priority levels).

Each of composition function defined by a different Hyper/Net composition attribute is implemented in the following away:

- Override: returns the method with the override attribute.

- Merge: renames the merged method and creates a new version of the method that invokes the original ones. It will keep the return results of calling each method and, finally, return the result of invoking a summary function with this set of results.

- Bracket: changes the original method so it first invokes the before method, runs the existing code, retaining its return value and invokes an after method with it. Finally, the after method is used to compute the return result.

If the methods being composed belong to a partial interface, then they will only be method declarations. Method declarations should not be merged or bracketed according to the previous composition functions because they would result in methods with bodies. This way, interface method composition with Hyper/Net should be limited to override composition.

The implementation details of each Hyper/Net composition function define the completeness constraints that are applied to each set of input methods. In the case of override composition, the method with the override attribute must be complete. With merge composition, all of the original methods must be complete, because they will be used in .NET compilation. With bracket composition, the original method as well as before and after methods must be complete as they will also be used for compilation. The methods generated by merge and bracket must also be complete, but this is guaranteed by the Hyper/Net implementation.

6.2.3 Model Limitations

The decomposition/composition power of Hyper/Net can be used to overcome some of the limitations with the partial types approach in supporting overlapping concerns. With Hyper/Net, concerns that overlap due to method units that belong to all of them can be avoided. This is done by decomposing these methods into the different concerns. These concerns may belong to any dimensions. Properties, variables and constructors are still not decomposable with Hyper/Net. Yet, as a work-around, methods can be used instead of variables and properties to achieve decomposition. Even though, Hyper/Net should offer composition support for other units below the class level, other than methods.

Decomposition will not always solve the issues with overlapping concerns. It is true that some dimensions contain decomposed units that would only overlap concerns in different dimensions if they were indecomposable. This is frequent with dimensions created to hold new kinds of decomposed units. For example, methods which provide discounts in a business rule dimension or methods providing logging in a non-functional requirements dimension will not exist in other dimensions, namely the Features dimension. Such dimensions are fully supported by a physical decomposition model. But, some dimensions simply provide an alternative view for existing units, namely the object dimension. In these dimensions, there is

no possible decomposition to avoid overlapping concerns with other dimensions. This kind of dimension needs a non-physical decomposition mechanism; that is a virtual matching mechanism. These kinds of dimensions were defined as virtual dimensions in Subsection 3.2.1. Virtual dimensions are not supported by Hyper/Net or the partial type approach, except for the object dimension, as explained in the previous section.

Also, as analysed in the previous section, declarative completeness may not be mandatory due to compiler error reporting mechanisms. Nevertheless, method decomposition also extends the declarative completeness support in MDSoC. Method declarations can be introduced using empty methods that are then overridden by methods in other concerns. References to variable or property units can be replaced with the use of methods. Only constructor declarations are impossible to introduce with the Hyper/Net approach. Unless a hyperslice references constructor units that it does not contain, it can be made declaratively complete by introducing method declarations or replacing property/variable references with method references[20].

As Hyper/Net does not introduce a non-physical decomposition mechanism, hyperslice reuse still depends on source control mechanisms or other mechanisms to avoid code copy. Hyperslice reuse is still limited by composition, in particular by the strict type matching imposed by partial types and method matching. By introducing method composition, Hyper/Net allows methods from a reused hyperslice to be composed with other methods, but the method signatures, name and containing partial type must match.

Hyper/Net outputs composed source code for a particular project. As hypermodules are equivalent to projects in this approach, the Hyper/Net output for a project could simply be used like a hyperslice in new compositions, offering hypermodule reuse. The problem is that Hyper/Net output does not contain any partial types[21]. In this approach, matching can only be done with partial types of the same type and methods, with the same signature, inside matching partial types. Even though hypermodules could be reused as hyperslices in new hyperspaces, there would be no way of matching their units to units in other hyperslices.

If Hyper/Net did not implement partial type composition, the resulting code could be used in new compositions, as it would still have the original partial types. But, the code resulting from Hyper/Net processing would not be equivalent to the result of a hypermodule. Instead, it would be equivalent to a single project hyperspace in the partial types approach. The resulting set of hyperslices could be reused but this would not be equivalent to a reuse mechanism for hypermodules.

There is a more effective work-around that would enable hypermodule composition with Hyper/Net. As Hyper/Net implements partial type composition itself, the .NET partial type composition model could be slightly changed, to allow the composition of normal types with a set of partial types of that type. This would be equivalent to assuming all type declarations are partial. With this change to Hyper/Net, it would be possible to match non-partial types in the Hyper/Net output of a project (a hypermodule) to other classes in a new project that included it. This work-around still has the heavy limitation of matching by type, invalidating composition of different hypermodules that were not developed with the purpose of being composed together.

Fully supporting hypermodule reuse, as defined in the MDSoC model, would require the separation of composition definitions from the code itself. It would also require the

[20] This change is intrusive to other hyperslices providing the referenced units. The introduction of composition constructs for variables and properties in Hyper/Net would overcome the need for this work-around.
[21] Hyper/Net also implements partial type composition (see Section 8.1 for details).

introduction of more powerful matching constructs than matching units by type or signature. Both partial types and Hyper/Net attributes belong to the code artefact. First, these composition constructs should be implemented in an artefact of their own, or in separate concerns of the code artefact. Then, constructs for matching classes and interfaces with different names/types and methods with different names should be introduced. This is naturally a case for the statement of future work for Hyper/Net (see Section 10.3).

As will be seen in Chapter 7 and Chapter 8, Hyper/Net supports input files in either VB.NET or C# and can output code in any of these languages, independently of the input language. If the issues with hypermodule reuse are overcome, Hyper/Net will immediately support hyperspaces with multiple languages. Projects in different languages could be composed together. Each project in a different language is a hypermodule in a different formalism. This way, each hypermodule can be processed, using Hyper/Net, to generate output in the same language. Then, the result of these hypermodules could be used together in a new project. This way, different formalisms (.NET languages) might co-exist in the same hyperspace. Furthermore, as Hyper/Net pre-processes .NET projects (before compilation), it could be changed to handle projects with input files in different languages. This is possible due to the features of the .NET language parser used by Hyper/Net that maps C# and VB.NET to the same Abstract Syntax Tree (AST) structure. The details on how Hyper/Net uses this AST are provided in Chapter 8. As seen, with the adequate changes, Hyper/Net could support multiple formalisms in the same hyperspace, something that no MDSoC implementation up to date is able to do.

As we saw, some limitations of the partial type MDSoC approach are overcome by Hyper/Net. But, because Hyper/Net is based on partial type matching, there are still some serious limitations:

- Hyper/Net supports overlapping concerns that can be removed through decomposition but not overlapping concerns introduced by virtual dimensions. Hyper/Net offers no virtual dimension support, except for the object dimension.

- Hypermodule reuse is still not possible. Hyperslice reuse is a little enhanced by the introduction of method composition. Nevertheless, it remains seriously limited by the strict matching model of partial types, not able to compose hyperslices developed without knowledge of each other.

- Finally, this approach is limited to a single artefact (code) and two formalisms (C# and VB.NET).

Some possible extensions to Hyper/Net that can overcome most of these limitations were already mentioned and are further developed in Section 10.3.

6.3 Conclusions

.NET partial types allow applying MDSoC without the need for specialized MDSoC software. The approach presented in Section 6.1 uses directories to support the hyperspace dimensions and concerns structure. Partial types are created inside the appropriate concern directory, implementing a physical decomposition of an MDSoC hyperspace. The .NET compilation process is in charge of composing the partial types together. Relying only in native .NET language features comes at the cost of severe reuse limitations. With the partial types

approach, hypermodule reuse is not possible and hyperslice reuse requires previous planning. Also, there is no virtual decomposition mechanism and, because there is no composition support for class members, these cannot be decomposed.

Hyper/Net adds support for method composition to the partial types approach. This allows methods to be physically decomposed into different partial types. Hyper/Net acts as a source code pre-processor. It is able to compose methods with the same signature using either merge or override composition. It also provides bracket composition that can be applied to any kind of methods. Even though Hyper/Net extends the partial types support for unit decomposition/composition, Hyper/Net lacks support for virtual decomposition. Virtual decomposition is required to implement particular types of dimensions that provide alternative views of a set of physically decomposed units.

If the reuse limitations of the partial types approach and Hyper/Net are overcome, there will be an additional benefit that also makes these solutions unique. Both approaches are able to support hyperspaces with multiple formalisms of the code artefact. With adequate reuse features it would be possible to compose units coming from different projects, written using different formalisms.

Chapter 7

Using Hyper/Net

This chapter presents guidelines on how to use .NET partial types and Hyper/Net for MDSoC. The first section describes how the partial types MDSoC approach can be implemented for .NET projects. Two usage scenarios are covered: how to apply this approach to existing code and how to use it initially, from scratch. The same scenarios are also covered in the second section, with the description of how Hyper/Net can be used when the partial types approach is not enough. Hyper/Net composition attributes are described in detail so the programmer is able to use them in different composition scenarios. The second section ends with the description of how to use Hyper/Net with .NET IDEs. This describes how the integration of Hyper/Net with existing programmer environments can be achieved.

The third section discusses how MDSoC projects can be tested. The focus is on unit tests. It is proposed that these tests share the same concerns of hyperspace as the tested code. Testing guidelines are presented for the two main .NET test platforms, with particular MDSoC approaches tailored for each.

Sections four and five present two complete examples of how MDSoC can be used to implement projects from scratch. The example described in the fourth section is limited to a single class so it can be more detailed. The example of the fifth section was used in most MDSoC literature and serves as a comparative and validation test for Hyper/Net. As the section shows, with Hyper/Net, it was possible to implement most of the example features in the way MDSoC literature presents them.

7.1 Using the Partial Types approach for MDSoC

This section succinctly presents the main usage lines of the partial types MDSoC approach. Even without using partial types, it is common to use the directory structure inside .NET projects to separate different features or concerns. This is done in an ad-hoc fashion and is limited to what the object decomposition allows to be separated.

The process of using .NET partial types for MDSoC was summarized in the beginning of Section 6.1. Figure 7 in that section provides a condensed view of this process. To create an MDSoC hyperspace in a .NET project, a programmer has to create a set of top level directories, one for each dimension. It is not uncommon for projects to have only a few top

level directories like this. Simpler projects will only have one. This structure is not definitive. New dimensions can be added latter and it is also easy to break up dimensions, merge dimensions or move particular contents between different dimensions. This is done simply using directory creation and drag-and-drop. These mechanisms are the supporting pillars of the growth and evolution of MDSoC projects.

A second directory level populates each of these dimensions with concern directories. These directories provide a finer granularity separation in a way that is defined by each dimension. Frequently, some dimensions have many more concern directories than others. Not all dimensions are equal in weight when compared. Yet, MDSoC allows the programmer to focus each of these dimensions and the concerns therein as if they we all equal. The only kind of directories in MDSoC hyperspaces that can contain files are concern directories. These files are usually code files but can also be resource files and other file types.

The two levels of directories in MDSoC hyperspaces can be all created initially or can be created as they are populated. The second approach provides the benefits of iterative development. Nevertheless, if prior to programming there was a solid MDSoC design phase, it is also adequate to initially create the entire hyperspace (directory structure) based on the dimensional structure of the design stage. It is also possible to define the dimensional structure of hyperspace as early as the analysis stage. This dimensional structure should then be used in the design and code artefacts.

Once there are concerns in the second level of directories, programming can start. Typically, any required types should be created as partial types in each concern. This way, the same types can readily be introduced in other concerns once they are needed there. The partial types in each concern will only contain the members that pertain to that concern. A .NET project that implements an MDSoC hyperspace can always be extended and changed. For instance, particular members of a type can be moved to the respective partial type in any other concern, even a new concern. If a particular concern starts to concentrate too many units it is also possible to decompose it into different concerns by moving the units (for example, class members) into a set of new concerns. It is also possible to merge different concerns by doing the inverse process, moving units in the different concerns into a single new one. This process consists mainly of copy-and-paste or even of drag-and-drop operations.

The versatility provided by .NET projects and the MDSoC hyperspace directory structure also allows direct support for mix-and-match operations. Creating a different version of a program, by removing a set of concerns, is as simple as removing the respective set of directories from the project. These can be reintroduced latter on. This allows using the same .NET MDSoC project to generate different flavours of the same application. This kind of functionality can be particularly useful when developing software product lines.

A normal .NET project can be converted into an MDSoC hyperspace implementation. This allows programmers to apply MDSoC to existing software. The units in the original code must be matched to concerns in a hyperspace. Again, hyperspace creation, using the two level directories structure, is the beginning of the process. Entire types can be matched to concerns, but usually only part of a type's members is matched to a concern. This is handled by creating the adequate partial type in the concern and moving in the type members that belong there. Applying MDSoC to existing software is a refactoring task, so the functionality of the software must not be changed during this process. At all times, it should be possible to compile the code and run any validation tests. These tests can be used to guarantee that the project functionality is not changed. Section 7.3 presents some of the particularities of testing .NET MDSoC projects.

During the process of applying MDSoC to an existing .NET project, decomposed code coexists with non-decomposed code. This violates the MDSoC model, where units must always belong to at least one concern. This is necessary during the refactoring task but it is not advised to leave only part of a project decomposed using MDSoC. Doing so may cause difficulty in understanding the project and it will not benefit completely from the advantages of MDSoC.

As presented in the previous paragraphs, the partial types MDSoC approach allows applying MDSoC to existing non-MDSoC projects, as well as developing MDSoC projects from scratch. The main requirement to apply MDSoC to existing projects is obvious: the source code for the project must be available. That code should also be editable so it can be decomposed.

It is important to notice that .NET partial types can be used with classes but also be with interfaces. This is important when the same interface should span different concerns and is achieved by decomposing the interface into partial interfaces in each of the appropriate concerns. A partial interface will only declare the methods that are important for that particular concern. These methods will then be implemented by classes that implement the interface in that concern.

In the partial types approach, members of a type that are declared in a specific concern are available throughout the remaining concerns. Furthermore, a type that does not exist in the concern the programmer is working on, can be initialized and used from it. This will require special care from the programmer. Once an element from another concern is referenced, a bound has been introduced. The element can be replaced with an equivalent one, eventually provided by a different concern. Still, changes to that element can be cross-cutting, as they may require changes to the way the element is used in the referencing concerns.

An important feature for developers in any programming paradigm is error reporting. This is a crucial functionality when developing with MDSoC. Adding to the standard error reporting for .NET languages, the MDSoC hyperspace structure itself should be validated, along with the composition rules and, eventually, any completeness constraints should be applied to their result. Regarding the MDSoC hyperspace structure, the MDSoC model does not allow units to exist outside concerns. But the partial types approach does and has no warning mechanism to help avoid these situations. If the MDSoC model was followed strictly, these situations should not be possible and would yield an error.

In the partial types approach, composition is defined by partial types and strict type matching. Partial type composition itself may not be possible to perform due to different kinds of errors. For instance, two partial classes of the same class type may extend a different class. This is not supported in common .NET languages. It will yield an error and the problem should be located in the partial classes that originate it (extend the different classes). Nevertheless, the Microsoft .NET 2.0 compiler does not identify the error at this location. Instead it locates it in the first partial class declaration for the class, which may even not extend any class. This forces programmers to search the remaining partial classes for the origin of the error. A more adequate localization would allow for immediate correction in the concerns of interest.

If a type resulting from composition does not meet the .NET completeness constraints, the compiler detects the missing elements and should trace them back to the original source code. For example, the missing elements can be methods that should exist because of a class implementing a particular interface. In this case the partial class that implements the interface should be located as the source of the error. Again, the Microsoft .NET 2.0 compiler does not do this and localizes the first partial class declaration.

Other kinds of errors, for instance syntax errors, are detected normally and reported in their locations inside concerns of the MDSoC hyperspace. Here the Microsoft .NET 2.0 compiler works in an adequate fashion from the MDSoC perspective. It takes programmers to the specific concern of interest and simplifies error solving with MDSoC.

The inadequate behaviour of the Microsoft .NET 2.0 compiler with composition and completeness errors should be corrected. If these errors were adequately localized, the programmer would be helped in solving the problems in the appropriate concern. For instance, with missing interface method implementations, these implementations should be done in the same partial type that implements the interface. This would be pointed out by an adequately localized error report. Nevertheless, for the remaining kinds of errors, the native .NET partial types approach reports errors in the adequate concerns, extending the advantages of using MDSoC to error correction.

7.2 Using Hyper/Net for MDSoC

When using the partial types approach for MDSoC, as classes are decomposed into different concerns, some of their methods can also match more than one concern. The partial types approach does not provide a way to decompose these methods into the appropriate concerns. Without this possibility, programmers have to rely on different workarounds, like creating differently named methods in each concern, which affects the code that uses the class in question, or keeping the method in only one of the concerns it belongs to. Neither workaround is satisfactory. This is where Hyper/Net comes into action. With Hyper/Net composition attributes these methods can be decomposed as necessary. Hyper/Net will be in charge of reuniting the decomposed methods according to composition attributes. The ability to decompose methods is useful when decomposing existing (indecomposed) code, refactoring an existing decomposition or by allowing the creation from scratch with decomposed methods.

A method should be separated into the appropriate number of smaller methods, each containing the part of the original method functionality (statements) that belongs to a specific concern. To introduce these methods into the appropriate concerns, they should be added to a partial class in the appropriate concern. The partial class must be of the same type of the class to which the original method belongs to. This partial class may already exist. If it does not, a new partial class for the type can be created to hold the method in the appropriate concern. The remaining units are similarly matched to a concern as they were with the partial types approach. The only difference here is that a method with the same signature can exist in several partial classes for the same class type. Also, like the partial types approach, Hyper/Net can be used with new MDSoC projects. In this case, the methods can already be created in a decomposed form inside the appropriate concerns.

Hyper/Net composition attributes are particular .NET attributes that can be applied to methods. Each Hyper/Net composition attribute provides a different way to compose methods. Hyper/Net offers the following composition attributes:

- *MethodMerge* – used for merge and override composition.
- *MethodBracket* – used for bracket composition.

Override is the simplest composition construct. It can only be applied to a single method from a set of matching methods. The method it is applied to will be output into the composed code while the others are simply removed. The override construct is defined as a *MethodMerge* attribute with a *MethodMergeAction* of *Override*, as exemplified in Listing 3.

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

Listing 3. Syntax of the override composition attribute.

Both override and merge composition are defined using the *MethodMerge* attribute. This has to do with the common implementation for both composition methods and is further explained in Subsection 8.2.1.

The merge composition construct allows embodying all of the decomposed methods functionality into a single composed method. The methods involved in merge composition must be methods with the same signature that belong to different partials of the same type. The composed method invokes each of these methods in turn. A *MethodMerge* attribute with a *MethodMergeAction* of *Merge* should be applied to each involved method, as exemplified in Listing 4.

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

Listing 4. Syntax of the merge composition attribute.

Each attribute should also define a different priority parameter (<*Priority*> in Listing 4). This allows Hyper/Net to determine the order of invocation of the original methods in the composed method body. The priority parameter is optional. When not defined for *MethodMerge* attributes with a *Merge MethodMergeAction*, the priority is set to the default value of -1. Because there must be a different priority for all merged methods, at most one method can lack an explicit priority definition. No method should define a priority of -1 explicitly. The default value may be particularly useful in the case of merging only two methods. The priority arguments define a total order in which merged methods will be executed.

Finally, a method that is used to compose the return values of the original methods can also be passed as an argument to the *MethodMerge* attribute (<*MergeResultMethod*> in Listing 4). This method must be local to the class to which the merged methods belong to. Two different *MethodMerge* attribute constructors allow defining this method, either by passing a typed delegate or by passing a string with the name of the method. The method must always have a particular signature (see Listing 5). The result composition method receives an array of objects that are the results of the different original methods and returns an object of its own, which will, in turn, be returned by the composed method. Currently, Hyper/Net only implements support for the second constructor using the string argument. In this case, the method is obtained at runtime and any failure to match the delegate type is only identified at that point. The typed alternative would be more adequate by providing compile time type validation.

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

Listing 5. The *MethodMergeResult* delegate type used by result merger methods.

Only one merge *MethodMerge* attribute in these methods should define a result merger method. If more than one is defined, Hyper/Net uses the last result merger method it finds; according to the order it processes the source code.

In merge composition, all merged methods should have one merge attribute. But there is an exception. One of the methods can have no such attribute if all the others have one. This case will be equivalent to that method having a merge attribute (a *MethodMerge* attribute with a *Merge MethodMergeAction*) with the default priority (-1).

The *MethodMerge* attribute can be used when an indefinite number of matching methods need to exist. Another composition attribute, *MethodBracket*, handles more specific scenarios, when one method must be preceded and succeeded by two other particular methods. The focus of the bracket composition is the method to be bracketed. With merge composition all methods involved are equally important.

```
[ MethodBracket(<BeforeMethod>, <AfterMethod>) ]
```

Listing 6. Syntax of the bracket composition attribute.

The method that will precede the method to be bracketed is the before method (*<BeforeMethod>* in Listing 6). This method receives information about the bracketed method and its arguments. Bracket composition must also define an after method (*<AfterMethod>* in Listing 6) which succeeds the bracketed method and determines the return value of the composed method. The after method receives the same information as the before method, along with the return result of the bracketed method.

Before and after methods will not have the same signature as the method to be bracketed. They must each implement a delegate type (see Listing 7) which provides the appropriate method signature with the required parameters and, for the after method, the *object* return type. Like with the result composition method in the merge construct, Hyper/Net only supports the identification of these methods using the *MethodBracket* constructor that passes their names as strings. Before and after methods must be local to the class that contains the bracketed method.

```
public delegate void BeforeMethod(MethodBase method, params object[]
                                  paramters);

public delegate object AfterMethod(MethodBase method, object returnValue,
                                   params object[] paramters);
```

Listing 7. Before and after methods implement delegate types.

In other MDSoC implementations either one of these methods is optional, but with Hyper/Net both the after and before methods have to be defined. This is imposed by the *MethodBracket* attribute constructor which must receive the identification of both methods. Nevertheless, one can always use empty before or after methods, which will not change the behaviour of the original method.

Merge composition can coexist with bracket composition. That is, a bracket attribute can be applied to a method that is already involved in a merge composition. Hyper/Net first processes the merge composition. The bracket is then processed with the merged method. This is equivalent to first generating the merged code and then applying the bracket attribute to the merged method. This way, with Hyper/Net, it is not possible to bracket a single instance of a method that is involved in a merge composition. The same does not apply to override and bracket composition. Override composition removes all matching methods but one. Any bracket composition attributes in methods other than the one with the override are also discarded. Bracketing is only applied in conjunction with override composition if it is applied to the same method as the override attribute. As for merge and override composition, these cannot coexist because they have contrary effects. Hyper/Net yields an error and stops

processing decomposed code when it finds merge and override attributes in the same set of matching methods. Finally, there can only be one composition attribute of each type applied to the same method inside a particular partial class.

As seen in the previous section, interfaces can be composed using the partial types MDSoC approach. Hyper/Net introduces the possibility of overriding or merging methods. This allows the implementation of interface methods to be decomposed into different concerns. The replication of the respective method declarations in different partial interfaces should also be possible to support the removal of particular concerns. For instance, an interface can declare a particular method in one concern and the method can be implemented in this and other concerns. If the concern where the interface declares the method is removed there will be no declaration for the interface method in the remaining concerns where the method is implemented.

Unlike method implementations in classes, method declarations in interfaces only define the signature of the method. As in Hyper/Net methods are matched by their signatures and the containing class type, there is no need to merge method declarations in interfaces. As such, override composition can be used with the method declaration, allowing it to exist in as many partial interfaces of the desired interface type as required. This is fully supported by Hyper/Net. Nevertheless, as the method declaration signatures must always be equal, the override attributes should not even be necessary. Supporting this only requires a simple change in Hyper/Net but is an issue for future work.

Merge and bracket attributes should not be used with interface method declarations. Nevertheless, Hyper/Net allows these attributes in method declarations and processes these declarations as if they were implementations. This will generate an invalid interface declaration in the composed code. It is also part of the future work plans for Hyper/Net to detected and disallow these situations.

In terms of support for existing code, Hyper/Net can be used to compose methods from separate existing projects. Still, there are some limitations. The code must be merged under a single project so that partial types can be merged[22]. The methods which can be merged must belong to partial classes for the same class type and have the same signature. These are very restrictive limitations. Still, if the original source code can be edited, it is possible to first decompose it and adapt it to an adequate hyperspace, where the methods that need to be composed match. Nevertheless, this approach is limiting from the perspective of reuse. By applying it, the code loses its original form. This means the resulting code may not be usable in place of the original code. Hyper/Net reuse limitations were already discussed in Subsection 6.2.3 and will be revisited in Section 9.2.

Statements inside a particular concern often need to refer to methods or other kinds of units in other concerns. More generally, concerns may depend on functionality from other concerns. Nevertheless, concerns should not depend on each other directly. Take, for instance, two concerns which provide the same functionality required by a third concern, the client concern. These two concerns may provide the same functionality through methods with different signatures. In this case, if the first concern is switched with the second one, even though the functionality is still present, the client concern will not be ready to use the functionality without being changed. This situation may be solved with appropriate composition constructs that operate at the level of method calls. Such composition constructs could be used to replace the method call with the call of an adapter that abstracted the different ways the same functionality is offered by different concerns. Introducing this kind of specialized composition

[22] This is also a limitation of the partial types approach with Hyper/Net.

constructs into Hyper/Net is an interesting issue for future work (see Section 10.3). Another solution might be to require that interfaces for inter-concern dependencies are defined and that the set of concerns which offer such functionality implement these interfaces. Nevertheless, what is important to keep in mind in the context of this section is that, with Hyper/Net, using functionality from different concerns introduces a dependency upon types and member signatures which Hyper/Net alone cannot help overcome.

On a more positive side, Hyper/Net method composition retains the ability of the partial types approach to perform mix-and-match operations. When the directory for a concern is removed from a project, the contained partial types and any decomposed methods therein are also removed. The inverse happens when a concern directory is reintroduced into the project. As merged methods must all have merge attributes, merge composition is adequately prepared for such changes during mix-and-match. It is not the case of override composition, as it may not be possible to remove a concern with an overriding method without changing the remaining code. This issue is further discussed in Subsection 10.3.1.

As for error reporting, Hyper/Net detects and reports composition and parsing errors. Only the first error detected is presented in the console output of Hyper/Net. These errors are relative to the decomposed code and can be analysed in the project source. For easier usage, an adequate IDE integration should present all detected errors and allow navigation to their locations. Other errors not detected during the composition or parsing phase will only be detected by the .NET compiler. These errors are located in the composed code that is output by Hyper/Net. This makes it more difficult for the developer to analyse and solve any such issues, having to understand the composition process. Future versions of Hyper/Net should address these error traceability issues and requirements.

7.2.1 Using Hyper/Net in SharpDevelop

A .NET MDSoC project using Hyper/Net can be developed using the SharpDevelop IDE like normal .NET projects can. The directory supported hyperspace is easily implemented using this IDE. The IDE also supports partial types and copes with repeated method declarations, so the look and feel of developing MDSoC projects should be pretty much the same as developing normal projects.

A .NET MDSoC project can only be compiled after it is processed by Hyper/Net. The project itself is only used by Hyper/Net and what needs to be compiled is the output file it generates. This can be achieved by calling the Hyper/Net console application (*HyperNet.exe*) with the appropriate arguments:

- The input project base directory, which is the path to the directory containing the .NET project.
- The input project file, which is the filename of the .NET project file. It can also be a relative path from the project base directory.
- The output file path, which is the path for the file where the composed code will be written. This can be a relative path from the directory Hyper/Net is invoked from.

The output file generated by Hyper/Net can be used as the single source code file of another .NET project which is then compiled normally. Hyper/Net will overwrite the file in the other project and the project can be built, finally generating the binaries for the original MDSoC project.

Instead of doing this process manually, it could be supported by build file. SharpDevelop supports the NAnt build system natively so it would be the build platform of choice for this purpose. Nevertheless, we did not explore this line of integration.

7.2.2 Using Hyper/Net in Visual Studio

Visual Studio has a similar behaviour to SharpDevelop in terms of .NET MDSoC project support. It also supports partial types and copes with repeated methods. This means the programmer can use standard IDE features, like Intellisense, in MDSoC projects.

The same integration described for SharpDevelop can be used with Visual Studio. Still, automatically invoking Hyper/Net as a project pre-compilation step is an alternative that we analysed using Visual Studio. The aim of this alternative is to allow compiling a .NET MDSoC project with a normal project build, making it transparent for the programmer that the project is using Hyper/Net, prior to .NET compilation, to be built. This requires that the build actions for all MDSoC source code files are changed from "Build" to "Embedded Resource". This is done so that the .NET compiler will not use these files during compilation. It will use the Hyper/Net output file instead. Then, a call to Hyper/Net is added as a project pre-build event. Visual Studio build macros, like *$(ProjectDir)*, can be used to provide the Hyper/Net arguments. A pretty generic example is:

D:\HyperNet\HyperNet.exe $(ProjectDir) $(ProjectFileName) $(ProjectDir)Output.cs

Here "*D:\HyperNet\HyperNet.exe*" provides the path to the Hyper/Net console application binary. The Hyper/Net arguments are obtained using macros. Only the output file has to be explicitly identified; the filename used is "*Output.cs*" inside the project root directory - *$(ProjectDir)*. To work with this model Hyper/Net was changed to only process code files which have an "Embedded Resource" build action. The build action of the Hyper/Net output file (*Output.cs* in the example) must be set to "Build". After this pre-compilation step generates the output code file, the compiler will build that code, eventually together with other source code files in the project that are not using MDSoC[23] and so, have a "Build" build action.

This solution meets our requirement of being transparent and supported by a standard build. Nevertheless, it has a major drawback. Most IDE features that help programmers (for instance, class diagrams and Intellisense) only make use of source code that is contained in files with a "Build" build action. This way, to be able to benefit from these features while developing MDSoC decomposed code, the build actions of all MDSoC source code files should only be changed to "Embedded Resource" before building the project. This is not practical at all. If Hyper/Net automated this change it would make this integration method the most transparent and adequate one. Nevertheless, to do this, Hyper/Net has to write a changed version of the project file. Visual Studio only processes these changes to the project file after compilation stars, working with the project version that existed prior to Hyper/Net pre-processing. We were not able to overcome this difficulty yet, so it is an issue for future work (also discussed in Section 10.3).

We also detected another issue that is related with the IDE behaviour to changes on loaded files. If the output code file is open in the IDE when the project is built, the compiler will use the output code file that existed prior to being written by the Hyper/Net pre-compilation step.

[23] Recall from earlier on in this section, that it is possible but not advised to have indecomposed code coexist with decomposed code in an MDSoC project.

This results in a build with a previous version of the composed source code. This way, the output code file should be closed when the project is built.

Finally, error reporting is not adequate with any IDE integration approach presented. In this approach with Visual Studio, the compilation fails when an error is detected by Hyper/Net. To analyse the error, the programmer has to check the "Output" tab in Visual Studio, where the error message is printed. Error reporting should identify the origin of the error directly in the decomposed source code. This error traceability requirement is another issue for future work.

This approach was developed with Visual Studio in mind but it also supported in Sharpdevelop. Like Visual Studio, SharpDevelop still does not support the enhancements proposed for an adequate integration, exhibiting the same behaviour when the project file is written in the pre-compilation step. Nevertheless, if the output code file is open when the project is built, SharpDevelop compiles the version of the file written by Hyper/Net and not the one in memory in the IDE. This way SharpDevelop does not require closing the output code file prior to compilation like Visual Studio does.

The support provided by Visual Studio and SharpDevelop is very similar. This subsection and the previous one were separated according to our integration experiences and not on final functionality. The details were presented in the context of each IDE where they were analysed more precisely. Then, each approach was also tested on the other IDE but without focusing on details.

7.3 Testing with MDSoC

Unit tests are a particular type of tests that address very specific code functionality. For example, a unit test can check method returns based on different inputs. Unit tests are usually implemented using code, frequently in the same language as the code which is being tested. Functional tests also relate with code, but focus problems from the higher-level perspective of requirements. Functional tests can also be created using the same language as the code they test. Both kinds of tests are related with the problems that are posed by requirements and are solved by code (among other artefacts). Functional tests address these problems directly and unit tests address the details of particular solutions to these problems.

In MDSoC, tests can define their own artefact or can share the code artefact with the application code. Both approaches are valid and feasible with Hyper/Net. In either case, tests address the same concerns as code does.

The considerations presented in the two previous sections can also be applied to tests that are implemented using source code. It is possible to decompose test methods into different concerns (using Hyper/Net method composition) or separate different test methods in a particular test class throughout these concerns (using partial types). It is also possible to remove specific concerns keeping the remaining concerns adequately tested.

We explore two unit testing frameworks in the context of MDSoC: NUnit and Microsoft Visual Studio Test Projects. NUnit [NUnit07] is an open source test framework. It supports the coexistence of tests and code in the same .NET project, so with NUnit, tests can share hyperslices with units from the code artefact. Nevertheless, tests are separated from the remaining code by having to follow particular restrictions. For instance, test methods have a

NUnit attribute to classify them as such (*[Test]*) and must belong to a class that is also classified with a particular NUnit attribute (*[TestFixture]*).

With NUnit, tests relative to a particular concern should be implemented inside the directory for that concern. Test methods must belong to test classes and thus exist separately from the remaining code in each concern. Test classes should usually be partial classes. Each partial class of the test class should belong to a different concern and hold the test methods belonging to that concern.

Methods that are composed using Hyper/Net should be tested from the perspective of each concern. If the methods are composed using override, the tests can be performed by separate test methods that are also composed using override. When two methods are involved, if the concern containing the overriding method is removed, the tests for that method are also removed. This way, the previously overridden method will now be part of the output code and will be tested by a test method in its concern that is not overridden anymore.

When merged methods must be tested, the task can become more complex and may require case by case analysis. Nevertheless, a general approach is still possible. Unit test methods should test the isolated functionality of the method in each concern. Because of the effects of composition these tests will possibly fail unless they are run while the containing concern is isolated from the other concerns. This is acceptable if we define that the purpose of unit tests is to test the local functionality of each concern at a low granularity. The results of composition can be tested with additional tests designed for a particular set of compositions. For instance, these can test the merged result of all methods or of particular combinations of these methods. Nevertheless, the most adequate solution would adapt itself to the different composition scenarios and cope with the removal and addition of concerns. Sometimes this is possible using MDSoC composition for test methods, but not all the times. This is an issue for future research.

Microsoft Visual Studio Test Projects were adapted from NUnit and are part of the Visual Studio IDE. Contrary to NUnit, Visual Studio only processes tests that exist in specific test projects, separate from the code projects. This is handled by creating the same directory structure in the test project that exists in the application project. This way, the test and application projects share the same hyperspace structure. The remaining approach described for NUnit also applies to Visual Studio Test Projects. Even the attributes that need to be applied to test methods and classes are similar.

Testing, in particular unit testing, is a valuable aid while creating projects using MDSoC. On top of the standard benefits of testing, it can be used to validate the composed behaviour and identify when local concern behaviour is affected by composition. To consolidate this unit testing approach with MDSoC, the following examples also describe how unit tests were applied to them.

7.4 Example: a Toll implementation

This example documents a simple development using MDSoC with .NET partial types and Hyper/Net. The example takes place in the context of the business of motorway operation, in particular tolled motorways. Some tolled motorways contain passing tolls where a fixed amount is charged. Other tolled motorways contain entry and exit tolls which are used to

calculate and charge a fare at the exit toll. This example pretends to represent an exit toll in the latter kind of motorway tolling system.

There are three main functional requirements for the exit toll that makes-up this example:

- Provide the necessary functionality to charge users based on the identification of an entry toll.
- Keep track of the number of vehicles that passed the toll.
- Finally, to help avoid congestions, an extra percentage should be charged during congestion periods.

This toll is implemented as a class library project that can be used as part of the software that manages the tolls. From that external perspective there needs only be a class implementing a method which receives the identification of an entry toll and returns the amount to be charged (see Figure 8). The other two requirements are considered to be internal to the project, so they do not need to expose properties or methods.

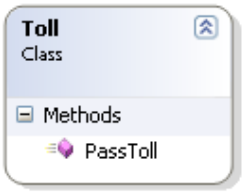

Figure 8. External perspective for the *Toll* class.

The three functional requirements are separated into different concerns of a Features dimension. This means there will be a *Features* directory in the project, containing one directory for each concern. The first requirement is implemented inside the directory for the Charging concern. The second requirement is implemented by counting passing vehicles and resides inside the directory for the Traffic Management concern. Finally, the third requirement is implemented inside the directory for the Congestion Charging concern.

The Charging concern contains a partial *Toll* class. It provides the *Toll* class with a method to calculate the amount that needs to be paid when arriving from any other toll (the *AmountFromOtherToll* method in Figure 9 and Listing 8). For simplicity purposes this method always returns the same value (5). The *PassToll* method uses the previous method to calculate the value which is returned, but also increments a local variable (*amountCharged*) containing the total amount received. Additionally this partial class provides a method to get the total amount ever charged for the toll (*TotalAmountCharged*). This method is used by the unit tests which are described further on.

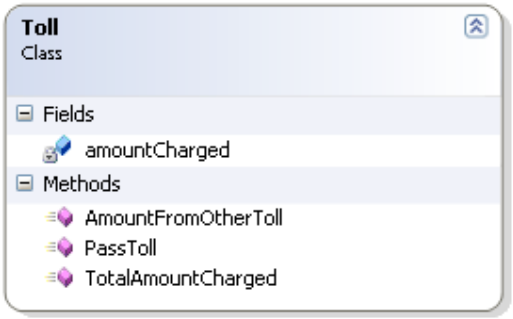

Figure 9. Class diagram for the partial *Toll* class inside the Charging concern.

```
public partial class Toll
{
	private int amountCharged = 0;

	public int TotalAmountCharged()
	{
		return amountCharged;
	}

	public int AmountFromOtherToll(Toll otherToll)
	{
		return 5;
	}

	public int PassToll(Toll originToll)
	{
		int amountToCharge = this.AmountFromOtherToll(originToll);
		amountCharged += amountToCharge;

		return amountToCharge;
	}
}
```

Listing 8. Partial *Toll* class implementing the charging requirement (Charging concern).

All the functionality that implements the charging requirement is located in this concern. Tracking of the total charged amount is very simplistic but it could evolve into a more complex functionality, requiring a concern of its own. In this case the Charging concern could be separated into two new smaller concerns.

The Traffic Management concern is also implemented as a partial *Toll* class. It also implements a *PassToll* method which simply increments a local integer variable (*numVehiclesPassedThisHour*) for each passing vehicle (see Figure 10 and Listing 9). This method can only coexist with its equivalent in the Charging concern if they are composed. This is done simply by applying a Hyper/Net *MethodMerge* attribute to the *PassToll* method. The attribute defines a *MethodMergeAction* of *Merge* so the functionality of this method is combined with the functionality of the same method in the Charging concern. The *PassToll* method in the Charging concern will have a default priority of -1. The explicitly declared priority of 0 for the method in the Traffic Management concern will make it run before the one in the Charging concern. The return value for the composed method will be calculated using an explicitly defined method: *SumInts*. As the *PassToll* method in the Traffic Management concern always returns 0, the composed method will return the same result as the Charging concern method. To keep up with the passage of time, a *PassHour* method has to be invoked. It simply clears the vehicle counter (*numVehiclesPassedThisHour* variable).

Finally, a *HourlyTraffic* integer property provides read access to the vehicle counter and is provided both for testing purposes as well as for use from the Congestion Charging concern.

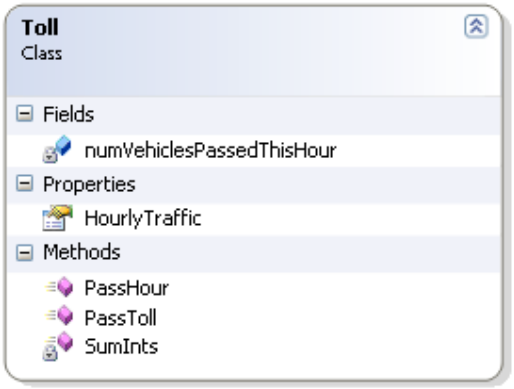

Figure 10. Class diagram for the partial *Toll* class inside the Traffic Management concern.

```
public partial class Toll
{
    private int numVehiclesPassedThisHour = 0;

      public int HourlyTraffic
      {
            get { return numVehiclesPassedThisHour; }
      }

      public void PassHour()
      {
            numVehiclesPassedThisHour = 0;
      }

    private int SumInts(params object[] ints)
    {
        int res = 0;
        foreach (int i in ints)
            res += i;
        return res;
    }

    [HyperNet.MethodMerge(HyperNet.MethodMergeAction.Merge, 0, "SumInts")]
    public int PassToll(Toll originToll)
    {
        numVehiclesPassedThisHour++;

        return 0;
    }
}
```

Listing 9. Partial *Toll* class implementing the vehicle counting requirement in the Traffic Management concern.

The Congestion Charging concern needs to apply an overcharge percentage to the return value of the *PassToll* method, but only in case a threshold value for the hourly traffic is reached. The most adequate composition mechanism for achieving this purpose is method bracketing. In particular, it is used to apply an after method to the return result of the *PassToll* method.

To be able to bracket the *PassToll* method, the method itself must also belong to this concern. This is achieved by introducing an empty *PassToll* method implementation (that simply returns 0) to which a merge attribute is applied (see Figure 11 and Listing 10). The priority of

this merge attribute is not relevant as the method has no side-effects and is only implemented to carry the bracket attribute. Nevertheless, the priority must be different from the priorities of this method in the other concerns.

The bracket attribute identifies an empty method (*Nil*) for before, simply because Hyper/Net bracket attributes must provide a before method. The implementation of congestion charging is achieved with the after method (*ApplyCongestionCharging*). Because method bracketing is applied after merging, the after method takes the return result of the composed *PassToll* method. It then uses the *HourlyTraffic* property, from the Traffic Management concern, to determine if a 100% congestion surcharge should be applied. This happens only when the hourly traffic surpasses 100 vehicles. In this case, the composed *PassToll* return value is multiplied by two. This calculates the value that is returned by the after method. It is also here that a local variable (*totalCongestionSurcharge*) is incremented with the surcharge amount. In case the hourly traffic is still below 100 vehicles, the composed *PassToll* return value is returned without change. By using the *HourlyTraffic* property directly this concern has a dependency on the Traffic Management concern. This situation has been discussed more generally in previous Sections (6.2 and 7.2).

Recall that in the Charging concern a *TotalAmountCharged* method returns the total value collected at a toll. By collecting a surcharge, the Congestion Charging concern is increasing the total value and also needs to complement the behaviour of the *TotalAmountCharged* method for correctness. This is done using merge (a *MethodMerge* attribute with a *MethodMergeAction* of *Merge*) on a local *TotalAmountCharged* method that returns the total surcharge value for the toll. The *MethodMerge* attribute defines that the composed result should be the sum of the merged method results by defining the *SumInts* as the result merging method. This way, the *TotalAmountCharged* method will return the normal charged value plus any eventual congestion surcharges.

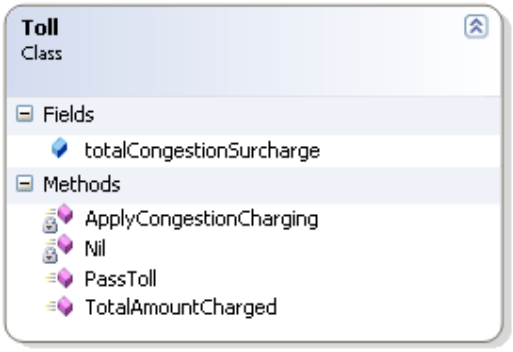

Figure 11. Class diagram for the partial *Toll* class inside the Congestion Charging concern.

```
public partial class Toll
{
    public int totalCongestionSurcharge = 0;

    private void Nil(MethodBase method, params object[] parameters)
    {
    }

    private object ApplyCongestionCharging(MethodBase method, object
returnValue, params object[] parameters)
    {
        if (this.HourlyTraffic > 100)
        {
            totalCongestionSurcharge += (int)returnValue * (2 - 1);
```

```
            return (int)returnValue * 2;
        }
        else
            return returnValue;
    }

    [HyperNet.MethodBracket(Nil, ApplyCongestionCharging)]
    [HyperNet.MethodMerge(HyperNet.MethodMergeAction.Merge, 2)]
    public int PassToll(Toll originToll)
    {
        return 0;
    }

    [HyperNet.MethodMerge(HyperNet.MethodMergeAction.Merge, 0, "SumInts")]
    public int TotalAmountCharged()
    {
        return totalCongestionSurcharge;
    }
}
```

Listing 10. Partial *Toll* class implementing the congestion charging requirement (Congestion Charging concern).

To validate the behaviour of the *Toll* class, a respective test project was also created. A separate project was required because we used the Microsoft Visual Studio unit testing framework (see Section 7.3 for details). This test project is also organized using the same MDSoC dimension and concern directories as the example itself. Test methods exist inside the three concern directories. Each test method implements a unit test as seen from the perspective of the concern. For instance, a test method in the Charging concern (*TestInitCharging* in Listing 11) checks if a newly created toll does not have any charged amount. Another test method in this concern (*TestCharging* in Listing 11) checks if the total charged amount is incremented by each vehicle passage.

```
[TestMethod]
public void TestInitCharging()
{
    Toll toll1 = new Toll();

    Assert.AreEqual(toll1.TotalAmountCharged(), 0);
}

[TestMethod]
public void TestCharging()
{
    Toll toll1 = new Toll();
    Toll toll2 = new Toll();

    toll1.PassToll(toll2);

    int charged1 = toll1.TotalAmountCharged();
    Assert.IsTrue(charged1 > 0);

    toll1.PassToll(toll2);

    int charged2 = toll1.TotalAmountCharged();
    Assert.IsTrue(charged2 > charged1);
}
```

Listing 11. Test methods in the Charging concern.

One of the Traffic Management concern test methods (*TestInitForTraffic*) checks if a newly created toll has no hourly traffic after initialization. Another test method in this concern

(*TestHourlyTrafficCounting*) simply tests if three consecutive vehicle passages are counted. Finally, another test method (*TestPassHour*) checks if, after the *PassHour* method is invoked, a toll that had counted vehicles is reset back to a zero hourly traffic value. The tests in this concern do not need any composition with the tests in the Charging concern because no result values of methods merged between these two concerns are affected by both concerns. The Charging concern determines the return value of the *PassToll* method while the Traffic Management concern only uses the *PassToll* method to increment its own counter.

```
[TestMethod]
public void TestInitForTraffic()
{
    Toll toll1 = new Toll();

    Assert.AreEqual(toll1.HourlyTraffic, 0);
}

[TestMethod]
public void TestHourlyTrafficCounting()
{
    Toll toll1 = new Toll();
    Toll toll2 = new Toll();

    toll1.PassToll(toll2);
    toll1.PassToll(toll2);
    toll1.PassToll(toll2);

    Assert.AreEqual(toll1.HourlyTraffic, 3);
}

[TestMethod]
public void TestPassHour()
{
    Toll toll1 = new Toll();
    Toll toll2 = new Toll();

    toll1.PassToll(toll2);
    toll1.PassToll(toll2);
    toll1.PassToll(toll2);

    Assert.AreEqual(toll1.HourlyTraffic, 3);

    toll1.PassHour();

    Assert.AreEqual(toll1.HourlyTraffic, 0);
}
```

Listing 12. Test methods in the Traffic Management concern.

The Congestion Charging concern contains a single test method that checks if passing a toll without congestion is cheaper than passing it when the congestion threshold has been surpassed.

```
[TestMethod]
public void TestCongestionCharging()
{
    Toll toll1 = new Toll();
    Toll toll2 = new Toll();

    int chargedWithoutCongestion = toll1.PassToll(toll2);
    Assert.IsTrue(chargedWithoutCongestion > 0);
```

```
    for (int i = 0; i < 200; i++)
    {
        toll1.PassToll(toll2);
    }

    int chargedWithCongestion = toll1.PassToll(toll2);
    Assert.IsTrue(chargedWithCongestion > 0);
    Assert.IsTrue(chargedWithCongestion > chargedWithoutCongestion);
}
```

Listing 13. Test method in the Congestion Charging concern.

The tests in the Charging and Congestion Charging concerns might use the actual price values returned by the *PassToll* method. In such case, some tests methods might also have to be composed, because the price values are affected through method compositions by these two concerns.

7.5 Example: the Expression SEE

Section 3.3 presented the expression SEE example that is used throughout most MDSoC literature. This example was also implemented using Hyper/Net. It was implemented according to the same approach as described in the original literature, in particular [Tarr01], which is the most detailed. Implementing the most relevant example from MDSoC literature using partial types and Hyper/Net is one form of validation of our MDSoC approach.

The requirements and design artefacts of the Expression SEE have already been addressed in [Tarr99] and [Tarr01]. The hyperspace for the expression SEE will have only two dimensions, the object and the Features dimensions. We will be working from the perspective of the Features dimension.

The Features dimension is materialized by a *Features* directory in a .NET class library project that implements the example. Inside, there is a directory for the each concern. One such directory contains the Kernel concern. It is a basis concern where each class is declared, related to others through inheritance and offers basic functionality like constructors. The classes in the Kernel concern also contain necessary private variables. Figure 12 provides the class diagram that depicts the class hierarchy of this example along with the Kernel concern methods and variables in each class.

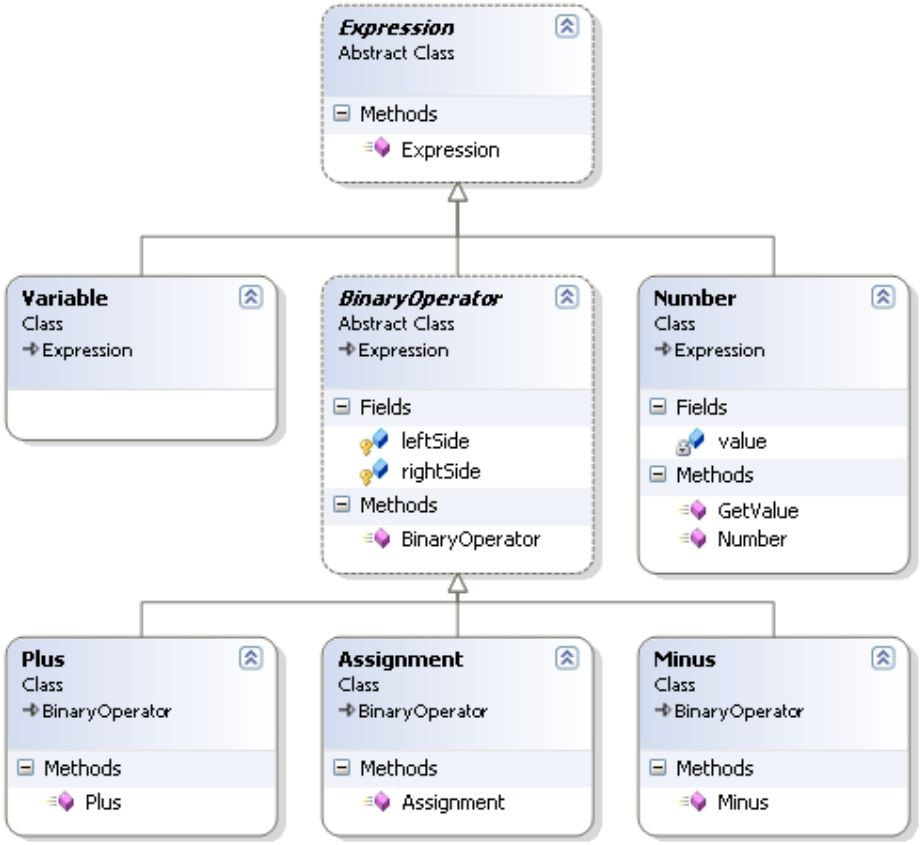

Figure 12. The class hierarchy in the Kernel concern of the Expression SEE example.

As defined in the original example, *Expression* is an abstract super-class for all the other classes. The Kernel concern defines the class hierarchy of this example, so, the remaining concerns do not need to define any inheritance relationships. *Number* and *Variable* classes extend *Expression* with the expected functionality. In the case of *Number*, it keeps track of its value, offers an accessor method[24] (*GetValue*) and a specific constructor. Binary operators share many characteristics, namely containing two different expressions, one on the right of the operator, another on its left. These are captured by the *BinaryOperator* class which derives directly from *Expression*. All of the three specific binary operator classes (*Plus*, *Minus* and *Assignment*) inherit their Kernel concern functionality from *BinaryOperator*.

All the classes declared in the Kernel directory (concern) are defined as partial classes so they can be further enriched in other concerns.

The Display concern focuses on printing expressions on the screen. This functionality is provided by a *Display* method in the classes of the expression hierarchy. This concern takes advantage of the partial types MDSoC approach. Each class of the hierarchy has a partial in this concern. These are represented in Figure 13.

[24] We use accessor methods instead of properties because Hyper/Net only allows the composition of methods and there might be a need to compose these accessors.

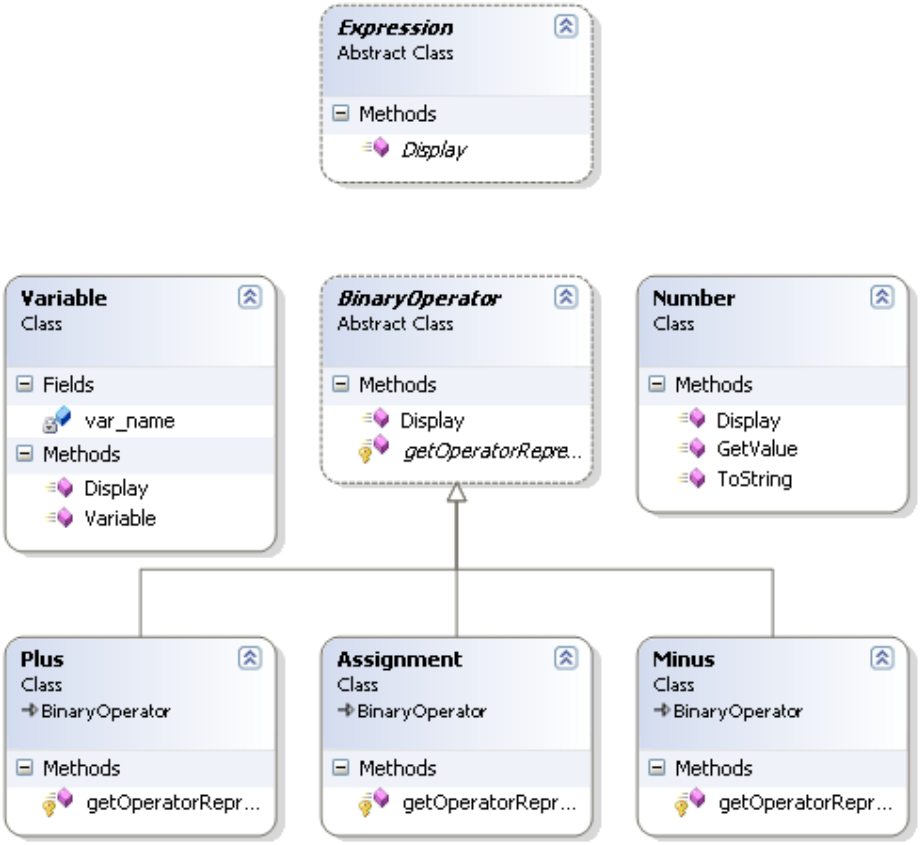

Figure 13. Class diagram for the Display concern.

Each of these partials implements a *Display* method, except for the *Plus*, *Minus* and *Assignment* classes. These use the *Display* implementation from their parent class (*BinaryOperator*) and only provide a specific helper method, *getOperandRepresentation*, that it requires. The *Display* method simply prints a representation for the expression on the screen. For example, for a *Number* object, it prints its integer value.

The partial classes in this concern can define no class hierarchy. Nevertheless, there is a dependency on the particular class hierarchy implemented in the Kernel concern. The *Plus*, *Minus* and *Assignment* classes rely on the Display method implementation from the *BinaryOperator* class. If *BinaryOperator* was removed as the parent of any of these three classes these would lack a *Display* method implementation. The need for particular parts of the class hierarchy in this concern can be expressed by defining inheritance only for the involved partial classes. In this case, by having the *Plus*, *Minus* and *Assignment* classes extend the *BinaryOperator* class. The remaining class hierarchy can be changed in the Kernel concern without affecting this concern. But, if these required inheritance relationships are changed in the Kernel concern, they will conflict with the inheritance directives in this concern and result in a partial type compilation error.

The Evaluation and Check concerns are implemented much in the same way as the Display concern. Each adds a new method to the classes, *Eval* and *Check* respectively (see Figure 14 and Figure 15).

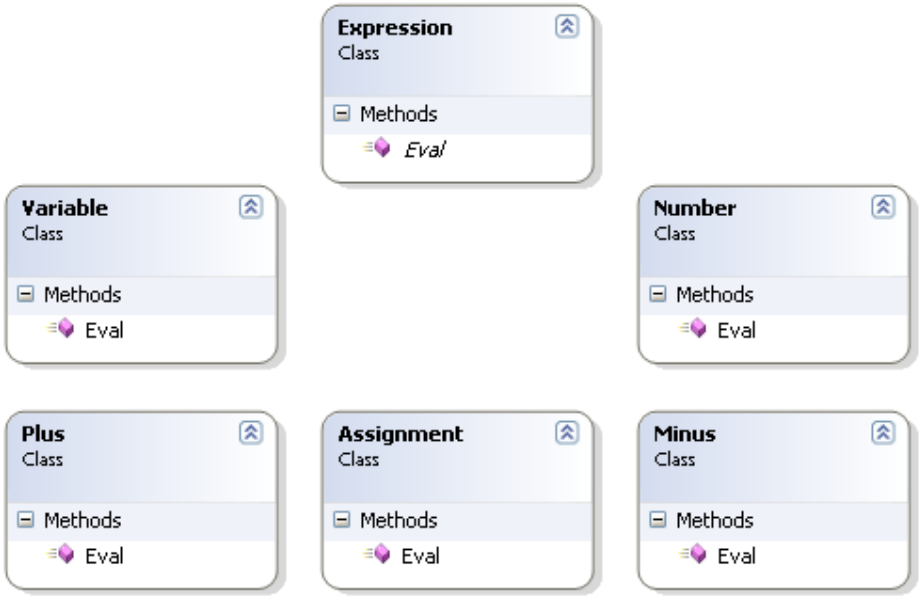

Figure 14. Class diagram for the Evaluation concern.

Not every class has a partial implementation in each concern. For example, there is no way to implement common binary operator evaluation. So the *Eval* method is only implemented in *BinaryOperator* child classes and there is no partial for the *BinaryOperator* class in the evaluation concern.

Another case occurs in the Check concern. Check functionality for binary operators is implemented in the parent class and inherited by the *Plus* and *Minus* child classes, without any additional implementations. The *Assignment* class overrides the binary operator *Check* method and implements an additional check because the left side expression must always be a variable, as only variables can be assigned to.

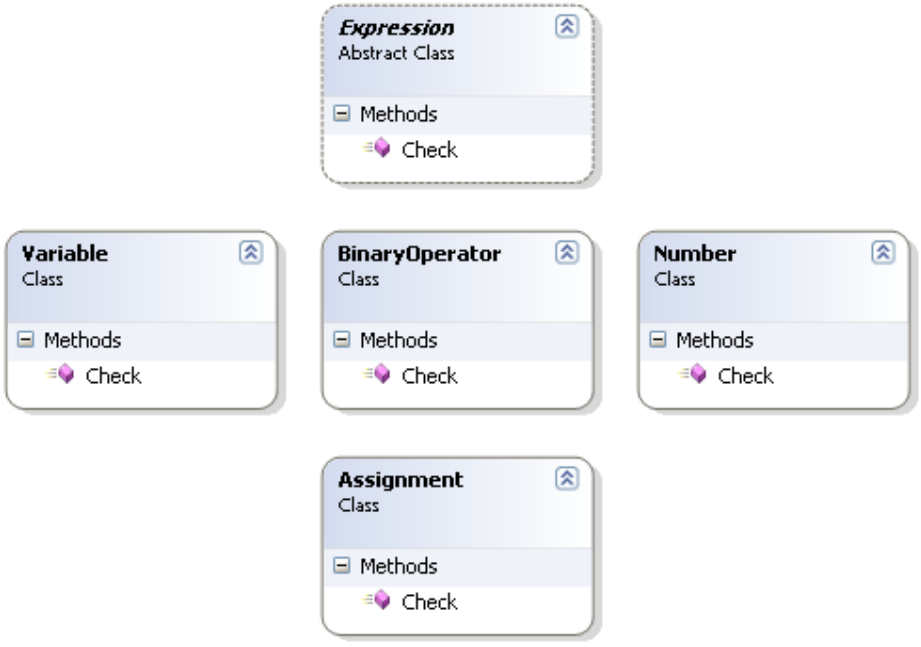

Figure 15. Class diagram for the Check concern.

Up to this point, this example did not require the use of Hyper/Net and was fully supported by the partial types approach. Also recall that up to this point the original example was easily implemented with OOP (see Section 3.3). The major advantages of using MDSoC here, in particular the partial types approach, is that the programmer is able to create and evolve each concern separately. Furthermore, it is possible to mix-and match concerns. To create a version of this project without either one of the Evaluation, Check or Display concerns it is only required to remove the respective directories from the project and build it.

[Tarr99] and [Tarr01] propose an extension to the Expression SEE that was addressed (in Section 3.3) by adding a Style Checking concern. This new concern should offer its functionality through the same *Check* method that was introduced in the Check concern. This enables existing code that uses expression checking to do style checking without needing to be changed.

At this point we find a major limitation with the partial types approach. 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. This happens because the *Check* method cannot be defined twice. Remember that each partial class is composed into a unique class in an additive fashion. All of the elements declared in the partial classes will belong to the resulting composed class, which cannot have duplicate method definitions. This is where Hyper/Net method composition is required.

A new Style Check concern is created, providing partial implementations for the *Check* method (see Figure 16). As a simplification of the original style check feature, this Style Check concern simply contains a check for the size of the name of variable elements which must be smaller than 5 characters. This way, this concern contains a default *Check* implementation for the *Expression* class, which always returns true. This default behaviour is overridden in the binary operator, to make sure the expressions on each side are correct. Finally, it is also overridden in the *Variable* class, to check if the size of its name is smaller than 5 characters.

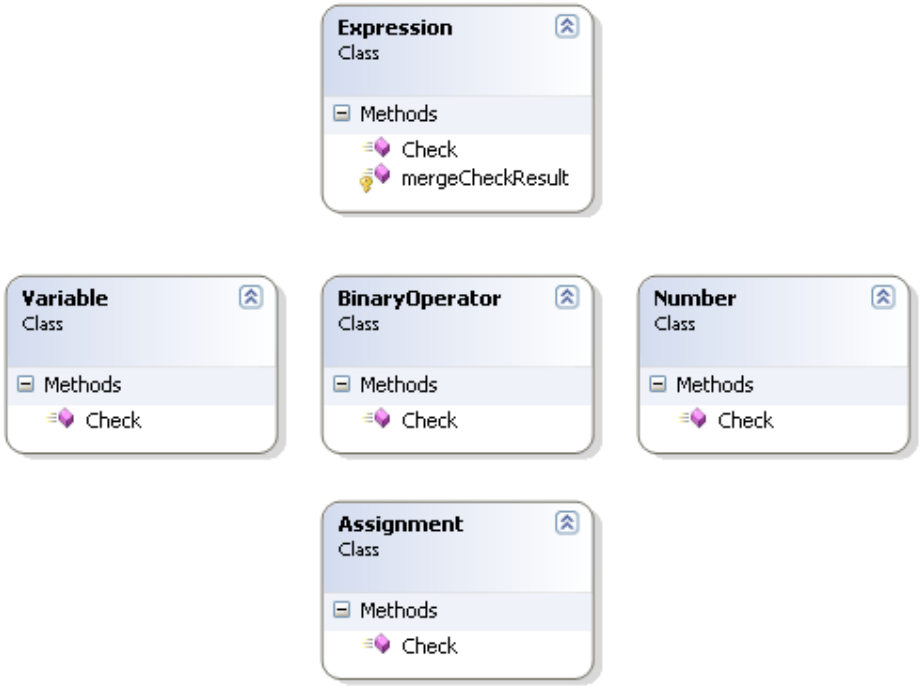

Figure 16. Class diagram for the Style Check concern.

Each of the *Check* methods in the Style Check concern has a *MethodMerge* Hyper/Net attribute declaration applied to it (see Listing 14). This defines how these methods are composed with their counterparts in the Check concern. The attributes define a method merge action of merge (instead of override) so the functionality of existing methods coexists with the new functionality implemented in this concern. The -10 priority level used, by being smaller than the -1 default priority, defines that these methods are invoked after the existing *Check* methods from the Check concern. Finally, the *MethodMerge* attribute also identifies a result merging method, *mergeCheckResult*. This method will only return true, if all the composed methods return true.

```
public partial class Variable
{
      [HyperNet.MethodMerge(HyperNet.MethodMergeAction.Merge, -10,
"mergeCheckResult")]
      public override bool Check()
      {
            return this.var_name.Length < 5;
      }
}
```

Listing 14. The *Variable* partial class in the Style Check concern.

The Style Check concern originally did not contain an explicit implementation of the *Check* method for the *Number* class. This has interesting consequences when this concern is composed with the existing Check concern, which explicitly defines a *Check* method for the *Number* class. When the two check concerns are composed together, the *Check* method defined in the Check concern, for the *Number* class, overrides the merged implementation provided by the *Expression* class. This way, the resulting *Check* method for the *Number* class is simply the *Check* method from the Check concern. This is not an issue for this particular example because both the merged *Check* method for the *Expression* class and the *Number*

Check method in the Check concern always return true. If the Style Check concern *Expression Check* method did not always return true, then there would be a composition issue. In such case, the Style Check concern should also implement a *Check* method for the *Number* class. To avoid such issues in the evolution of the Style Check concern we already implemented the *Check* method for the *Number* class. This is an example of an important MDSoC design need. When relying on inherited behaviour for a specific concern, it is necessary to check if the inherited functionalities are not overridden in other concerns.

In terms of implementation, the style check feature rules from the original example could be implemented in much the same way as we implemented the check for the variable name length. But, by doing so, this example would only become more complex, without adding any new composition or usage scenarios.

Another feature for the Expression SEE, that can exemplify more Hyper/Net composition attributes, is logging method entries and exits for all method calls. This feature was proposed and exemplified (using Hyper/J) by [Tarr01]. [Tarr01] uses the bracket composition to trigger the invocation of logging for the entry and exit of all methods in the expression class hierarchy. This feature is also implemented with Hyper/Net using bracket composition.

First, a new Logging concern is introduced. This concern contains a partial class definition for the *Expression* class (see Figure 17). This partial class introduces two new methods that will be used for logging: *methodEntryLog* and *methodReturnLog*. These are protected static methods that respectively implement the Hyper/Net delegate signatures for before (*BeforeMethod*) and after (*AfterMethod*) methods. Static methods are lighter and more appropriate for the logging task than instance methods. But Hyper/Net bracket composition requires that the before and after methods are available as class instance members. This is why the *entryMet* and *exitMet* method delegates are introduced. They simply work as method reference holders for the static methods, making them available for invocation as class instance members. Finally, the *GetLogPrefix* is only a helper method used by both logging methods to write a logging prefix with the current date, time and the current thread identifier.

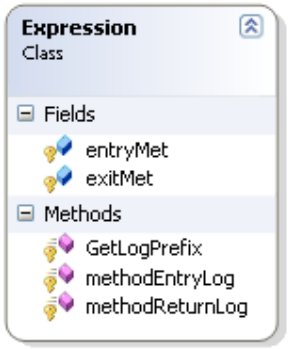

Figure 17. Class diagram for the Expression class in the Logging concern.

Relying on the inheritance hierarchy defined by the Kernel concern the logging methods become available to all other classes. These methods can then be used with a Hyper/Net bracket attribute to apply logging to the methods in each class (see Listing 15 for an example). This bracket attribute has to be explicitly applied to all the methods in all the classes of the hierarchy. Furthermore, to be able to declare the bracket attribute, there must be a declaration of these methods in the partial classes of the Logging concern. Each of these method declarations must be composed with the respective methods in the other concerns using

appropriate merge composition attributes. Listing 15 shows the two composition attributes used to bracket the *Check* method in the *Assignment* class.

```
[HyperNet.MethodMerge(HyperNet.MethodMergeAction.Merge, -20)]
[HyperNet.MethodBracket("entryMet", "exitMet")]
public override bool Check()
{
    return true;
}
```

Listing 15. Bracket **a**ttribute declaration for the logging feature.

Hyper/J uses a more powerful matching mechanism (based on regular expressions) to define, with a single composition statement, that all the methods in all the classes are bracketed. Achieving such expressivity with Hyper/Net is an issue for future work (see Section 10.3).

Another feature proposed for this SEE in [Ossher99] is caching. Each expression should cache the result of evaluation for future usage. Cache invalidation would also be an issue for this concern. This could be implemented if the *Eval* method was bracketed with methods such that the cache contents were tested to check if they were usable. If so, instead of evaluating the expression, this result should be returned. Yet, this would require an enhancement to Hyper/Net's bracket attribute to provide around functionality (another issue for future work).

The Expression SEE example was also tested using the MDSoC unit testing approach presented previously in Section 7.3. This was done using a Microsoft Visual Studio .NET test project that is organized using the same hyperspace dimensions and concerns as the example itself. In terms of the object dimension of tests, there is a single test class that implements different test methods and a test initialization method. Independent concerns like Display and Evaluation implement their own specific test methods in a partial test class. There are no conflicts as these test methods are also independent.

To test concerns that have methods which are composed together, like the check concerns, the test project also relies on Hyper/Net method composition. The tests for the Style Check concern allow the assignment of two binary operator expressions. But this is not allowed in the tests for the Check concern. A *TestCheck_assign_two_binOp* test method does this test accordingly in each concern. When the tests for each concern are run separately, by removing the other check concern from the tested and test projects, all unit tests pass. To test the composed functionality of both check concerns appropriately the *TestCheck_assign_two_binOp* test method has to be composed itself. The result of merging the functionality of the Check concern *Check* methods is that the strictest checks always apply. So, in this case, the assignment of two binary operator expressions is invalid. This is tested by applying an override composition attribute to the *TestCheck_assign_two_binOp* test method in the Check concern, which is the less strict test.

Hyper/Net composition is further used in the test project to merge test initialization methods that exist in different concerns. In this case, there was need to initialize two additional variables in the Style Check concern. This is done using the same test class initialization method signature, which has to be composed with the initialization method in the Kernel concern. These methods are composed using a merge composition attribute that is applied to the test class initialization method in the Style Check concern.

Finally, when a concern is removed from the tested project it also has to be removed from the test project. A different test approach that is possible using NUnit (also presented in Section

7.3) allows tests to coexist with code in the tested project. That approach has the advantage of simultaneously removing the code and tests when a concern directory is removed.

7.6 Conclusions

MDSoC hyperspace dimensions and concerns are supported on a very simple directory structure. This structure can easily be changed by moving files between concern directories. .NET partial types are used to place class members and other units into the appropriate concern directories. Additionally, Hyper/Net provides override, merge and bracket method composition using two different attributes that are applied to methods: *MethodMerge*, for overriding and merging, and *MethodBracket*, for bracketing methods.

Both the .NET partial types approach and Hyper/Net are versatile and allow creating hyperspaces from scratch or refactoring existing projects into hyperspaces. As for error reporting, with partial types, the .NET compiler allows tracing errors back to specific points inside the adequate MDSoC concerns. Hyper/Net is more limited in terms of error reporting. It is possible to develop normally with .NET IDEs using the proposed approaches. Still, Hyper/Net has some specific IDE integration issues that should be addressed as part of future work. When using any of the two approaches, the developer should take special care while introducing inter-concern dependencies. Normal .NET compilation and IDEs will also help in this regard.

.NET applications developed using MDSoC can be tested like other .NET applications. Tests address the same concerns as code does. So, tests implemented using code can be part of an MDSoC hyperspace created using Hyper/Net. These tests benefit from the advantages of MDSoC and provide testing benefits that are specific to MDSoC, like being able to test local concern functionality at low granularities and supporting mix-and-match.

Finally, using a very simple example, centred on motorway tolls, we provide a realistic usage scenario for merge and bracket composition and also exemplify dependencies between concerns. The SEE Expression example is a bit more complex, with its own class hierarchy, and provides an important evaluation scenario for Hyper/Net as it was also used as an example for Hyper/J. This example provides more diversified usage scenarios of all Hyper/Net composition attributes.

Chapter 8

Hyper/Net implementation

Hyper/Net is only a prototype implementation. It was created with the purpose of overcoming some limitations of the partial types native MDSoC approach. These limitations were identified while we were implementing the two MDSoC examples described in Sections 7.4 and 7.5 using only partial types. Hyper/Net successfully overcomes these limitations, as seen in Chapter 7. Implementing and using Hyper/Net also served as a mean to deepen our intimacy with MDSoC usage with .NET programming.

There were no analysis or design stages prior to Hyper/Net implementation. This chapter provides some analysis and design material from a reverse-engineering perspective. Refining this analysis is an issue for future work. One case that requires such refinement will be identified when future versions of Hyper/Net separate composition metadata from the code itself. In our current approach, composition meta-data processing and extraction is not separated from composition execution. If these two concerns had been separated from the start it would be easier to evolve Hyper/Net with this future work requirement. This is just an example of how Hyper/Net code may be inadequately organized. Furthermore, Hyper/Net it is not optimized or even fully tested.

Yet, we acknowledge the value of the Hyper/Net implementation. So, this chapter provides an MDSoC hyperspace analysis of Hyper/Net, drilling down to the details allowed by each dimension. This task is simplified as Hyper/Net is itself implemented using a partial types MDSoC hyperspace. We emphasize that this hyperspace was not carefully designed and originated as an organization attempt of our early Hyper/Net plain OOP implementation. This should not be used as a reference MDSoC hyperspace. Still, the analysis made of this hyperspace is itself an interesting approach. The versatility of the MDSoC perspective simplifies the documentation process, allowing details about the implementation to be provided in smaller, partly independent groups contained in hyperslices.

The first section presents a procedural perspective of Hyper/Net. This is equivalent to the analysis of a time dimension, yet it has no physical implementation with Hyper/Net. The second section focuses the physical architecture of Hyper/Net. It starts by detailing the MDSoC hyperspace implemented by Hyper/Net source code. The requirements realized by Hyper/Net are presented in the context of each concern of this hyperspace. The second section goes on to present the two separate projects that together implement Hyper/Net. One is a class library that needs to be included in .NET MDSoC projects that use Hyper/Net. This class library is analysed in detail from the perspective of its two concerns. The other one, the

Hyper/Net console application, is first presented from the perspective of the object dimension. This provides an overview of the code and a classic design presentation. Then, the details about each element of the application are provided from the perspective of functional dimensions. Here, each object is analysed in detail but only for a particular concern.

8.1 The process

As seen in Section 6.2, Hyper/Net works as a pre-compilation tool that processes source code. This way, Hyper/Net works before the compiler, transforming source code. This process is depicted in Figure 18 and described in detail below.

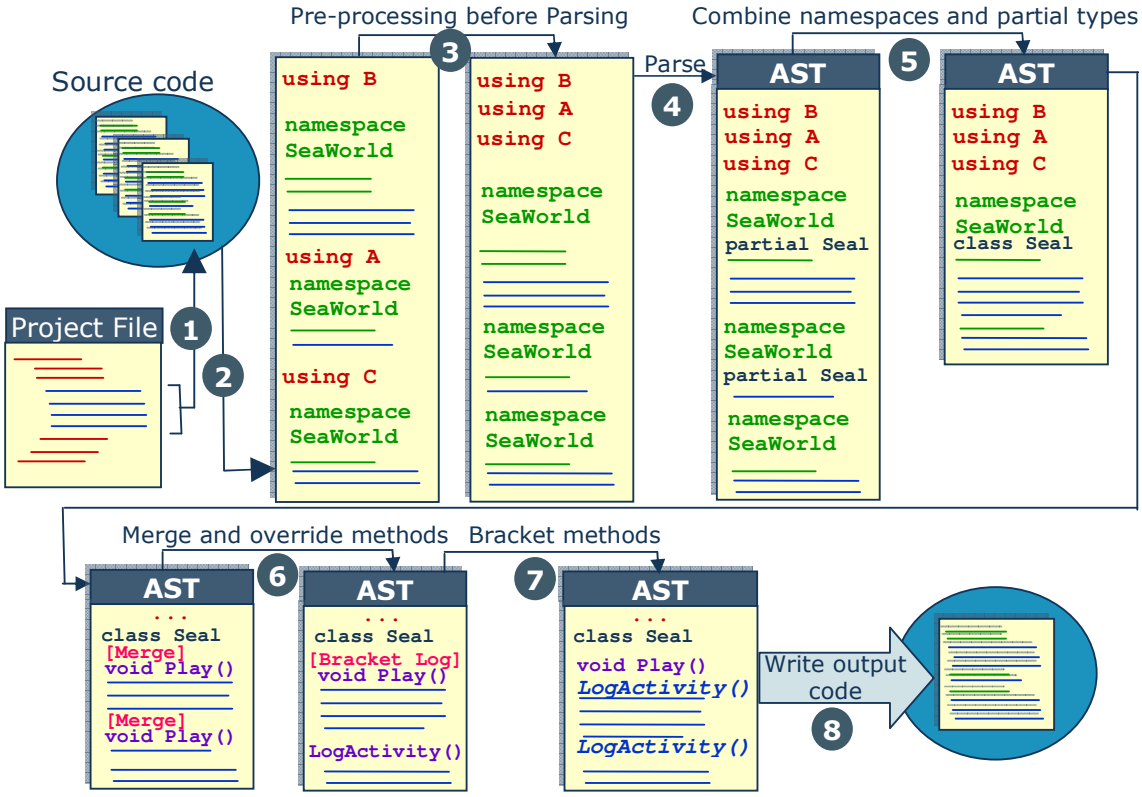

Figure 18. Illustration of a procedural view of Hyper/Net.
Hyper/Net starts by processing a project file to load source code files (1). It ends by writing a single composed source code file, ready for .NET compilation (8).

Hyper/Net receives as input an MSBuild project file[25]. It uses the project file to identify source code files that need to be processed by Hyper/Net prior to .NET compilation (see Figure 18, Step 1). These files are identified in the project as embedded resources, instead of compilation resources. This is how Hyper/Net distinguishes files it should process from files directly destined for the compiler.

[25] MSBuild project files are build-files, written in XML. Among other things, they contain information regarding the source code files that are involved in the project. For instance, it identifies the files that need to be compiled to produce binary output for the project. These are marked as compilation resources.

The identified source code files are read and their contents concatenated into a single body of source code (see Figure 18, Step 2). This is equivalent to reading the source code from a single file, except that a few ordering impositions of .NET languages are not maintained. For instance, source code files in .NET languages must have all using/import directives at the begging of the file. To abide to this rule, and support parsing the code just read, there must be an additional pre-processing step. It consists of moving all using/import directives to the beginning of the code (see Figure 18, Step 3).

Having made the body of source code valid, Hyper/Net then uses the NRefactory parser to produce an AST (abstract syntax tree) for the code (see Figure 18, Step 4). This parser produces the same AST structure from C# and VB.NET code. This way Hyper/Net supports source code in any of these two languages. After this step, all processing is done using the AST instead of the textual format of the source code. This is depicted in Figure 18 through the "AST" labelled box on top of the code blocks of each stage.

Following the parsing stage there is a composition preparation stage (see Figure 18, Step 5). It is in this stage that Hyper/Net merges partial types into a single type declaration. Before partial types can be merged, a similar merge has to be done with namespaces. Recall that the parsed source code is a concatenation of source code from different files. With the exception that using/import directives have been moved to the beginning of the code body. The same namespace can occur time and again in the concatenated source code. This is valid in .NET languages, but makes partial type merging more difficult as partial types would have to be matched across a wider scope. This way, the contents of different nodes for the same namespace are brought together into a single namespace node. This step makes partial types merging easier, as partial types can be searched for under the same namespace AST node[26]. Partial types are fully merged. This means that type references (inheritance and implementation of interfaces), attribute declarations and all partial type member declarations are brought together. This step has a purely additive nature and is limited to the class and interface level. Composition between class members is done in the next step. Both composition preparation procedures (merging namespaces and partial types) involve iterating through the AST.

The next stages of Hyper/Net processing are the core of our work. Merge and override composition is done in the same step (see Figure 18, Step 6). The AST is iterated through once again, this time searching for repeated instances of the same method. Methods are matched by signature, that is, the name, parameter types and return type must all match. As partial types were previously merged, the search for matching methods is local to each class node. At this point, any matching methods are searched for Hyper/Net attributes 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 on renaming the existing methods and creating a new 'super-method', the merged method, that will call each of the previous methods. The priorities defined in the merge attributes that are applied to each method are used for determining the order by which the methods are invoked. The result of each invocation will be kept in a local variable inside the merged method. 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 the list of results of each method invocation and returns only one value. Its return value is returned by the merged method. The result merging method can be defined as an argument of any of the merge composition attributes.

[26] Recall that partial types cannot span different namespaces. Two equal partial type signatures in different namespaces are considered as different types.

Bracket composition searches the AST directly for bracket attributes in methods (see Figure 18, Step 7). Once a bracket attribute is found, the method starts to be changed. First, the before method is invoked from the beginning of the existing method. The before method receives as arguments the original method meta-information and arguments. Then, prior to the original method return statement, the after method is invoked. The after method receives the same information as the before method and also the return result of the original method body. Existing return statements are replaced by the return of the after method. When the return statement expressions are replaced by the after method invocation, these expressions are passed as an argument to the after method.

At this point the code in the AST is ready for compilation. This way, Hyper/Net outputs the composed code in the language of choice (see Figure 18, Step 8), either C# or VB.NET. The code is output as a single source code file which can be compiled.

To compile Hyper/Net composed code it is not necessary to use a .NET 2.0 compiler. Instead, a .NET 1.0 or 1.1 compiler can be used. This happens because the partial types defined in the source code are processed internally by Hyper/Net, after the parsing stage, 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 code resulting from Hyper/Net composition needs not be aware of these partial types, because Hyper/Net already transformed them into complete types.

8.2 Hyper/Net internal architecture

Hyper/Net has itself been implemented using the partial type MDSoC approach presented in Section 6.1. Hyper/Net works as a command line tool, taking as argument a project file, which identifies the source code to be composed, and the output file path to write the composed code to. The command line tool is implemented using a Windows console application .NET project written in C#. Prior to using the command line tool, MDSoC projects that use Hyper/Net may need to use composition attributes. These attributes are declared in a different .NET project which is a class library .NET project, also in C#. This project needs to be referenced by any project which use Hyper/Net composition attributes. The Hyper/Net console application project also depends on the attribute class library and so, it also references it.

The hyperspace defined by Hyper/Net has concerns that are divided between these two different projects. Furthermore, the two different architectural concerns that are separated by these projects can be seen as populating a project dimension in the Hyper/Net hyperspace.

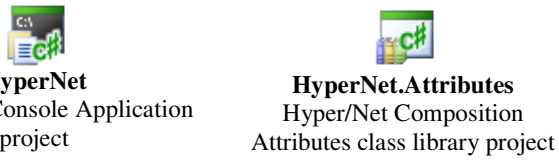

Figure 19. Hyper/Net viewed from the project dimension perspective.

Up to this point two Hyper/Net MDSoC dimensions have been presented. The first is the time dimension with the procedural perspective presented in the previous section. This dimension does not have a physical implementation. On the contrary, the project dimension depicted in

Figure 19, which is the second dimension, is at the top level of the physical decomposition of Hyper/Net.

These two projects were internally modularized using the partial types MDSoC approach. The hyperspace structure implemented in these two projects is represented in Figure 20. Along with the Object dimension, it is made up of two functional dimensions:

- Features: This dimension contains all concerns that are not specific to the composition attributes introduced by Hyper/Net. It contains general features like input/output handling, parsing and some related features.

- Language Features: The second dimension is populated by concerns that are specific to the language elements introduced by Hyper/Net. It contains specific features regarding Hyper/Net attribute-based language extensions. It is the only dimension present in the attributes project where it contains declarations for these concerns. In the console application project, these concerns are populated with the implementation that processes the respective Hyper/Net composition language features.

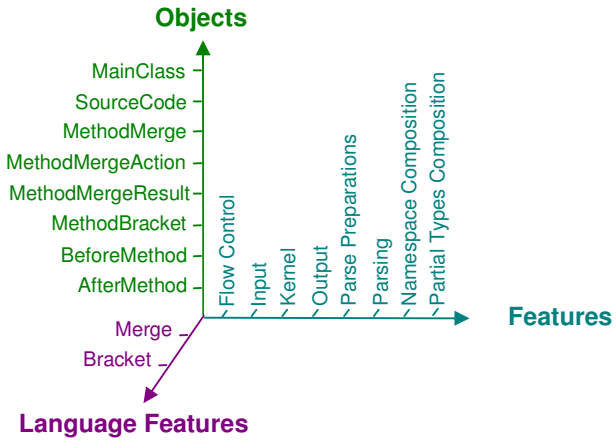

Figure 20. Representation of the MDSoC hyperspace used for Hyper/Net.

We now look at the concerns of the Features dimension, one by one, presenting the requirements addressed by each concern:

- Flow Control
    - Handles and validates command line arguments.
    - Invokes different processing steps in the appropriate order. Processing steps are factored into other concerns in this and the Language Features dimension.

- Input
    - Reads project files.
    - Processes project files to determine source code files.
    - Reads the source code files.

- Kernel

- Contains the core fields of each class and appropriate manipulation methods and constructors.

- Output
  - Provides a way to get source code that is held in memory in string format or is generated from a parsed AST.
  - Is able to write the required output files when invoked by other concerns. For instance, it is used to write the final composed code file and temporary code files used in debugging scenarios.

- Parse Preparations
  - Moves namespace import/using directives from the middle of a textual body of code to its top.
  - Should eventually do other preparations needed to support parsing the results of merging a set of different code files into a unique string.

- Parsing
  - Can parse code in a textual form. Generates an AST that can be manipulated programmatically.

- Namespace Composition
  - Provides a way to merge all units in an AST, which belong to the same namespace, under the same namespace node.

- Partial Type Composition
  - Provides partial type merging. This implements partial type composition like the .NET compiler, so it only does additive merging at class and interface level. After this, there can be repeated methods inside classes. This is an issue for the Merge concern in the Language Features dimension.

The Language Features dimension only contains two concerns but these address several related requirements:

- Bracket
  - Provides attributes that can be used to declare methods which should be bracketed in MDSoC programs.
  - Searches an AST for methods that should be bracketed.
  - Extracts bracketing information from these methods.
  - Uses bracketing information to apply bracketing, transforming the AST.

- Merge
  - Provides attributes that can be used to declare methods which should be merged or override others in MDSoC programs.

- Searches an AST for methods that should be merged or override others.
- Extracts merging information from these methods.
- Searches for all methods that match a particular method which should be merged or override others.
- Determines if the merging information is valid for a set of matching methods.
- Uses the merging information to merge the methods involved or have a method override all the others. This is done over the AST, changing the AST.

Curiously, if the Language Features dimension is removed, Hyper/Net implements a pre-compilation tool that merges partial types into complete types. It would be possible to use such tool to support partial types with a .NET 1.x compiler. This is an example of the capabilities of mix-and-match that are made possible by MDSoC. These have also been address as part of the examples of Chapter 7.

8.2.1 Composition attributes class library

This class library implements the first requirement presented for the Bracket and Merge concerns. For the Bracket concern, it is implemented with the declaration of the *MethodBracket* attribute and, for the Merge concern, with the declaration of the *MethodMerge* attribute. These are the attribute types that can be applied to compose methods in .NET MDSoC projects. By populating only these two concerns, this project exists only in the Language Features dimension of the Hyper/Net hyperspace. This is due to the fact that all the remaining features are already defined as part of the .NET languages so do not need to have any specific declarations provided by Hyper/Net.

Initially, these two attributes were declared inside the Hyper/Net console application project. This meant that each MDSoC project created using Hyper/Net would have to reference the entire Hyper/Net application. This class library was created as a refinement to this initial solution. Now, the class library contains exclusively the Hyper/Net code units that are publicly required and is the only binary that needs to be referenced from .NET MDSoC projects.

The two Language Feature dimension concerns in this project are physically decomposed. This way it is trivial to analyse the implementation of these concerns in their decomposed form. The code for each concern is already isolated, so it can be directly analysed without leaving the context of a particular concern. It is also possible to focus part of the design artefact of each concern using a class diagram depicting a single concern. Such a class diagram can be obtained directly in an IDE like Visual Studio by removing the remaining concerns from the project and generating a class diagram.

This project is analysed for the design and code artefacts in this fashion. The same approach will be taken for the concerns implemented in the Hyper/Net console application project in the following subsection.

Language Features – Bracket

The Bracket concern defines the *MethodBracket* attribute (see Figure 21). It is used to declare bracket composition for a particular method it is applied to.

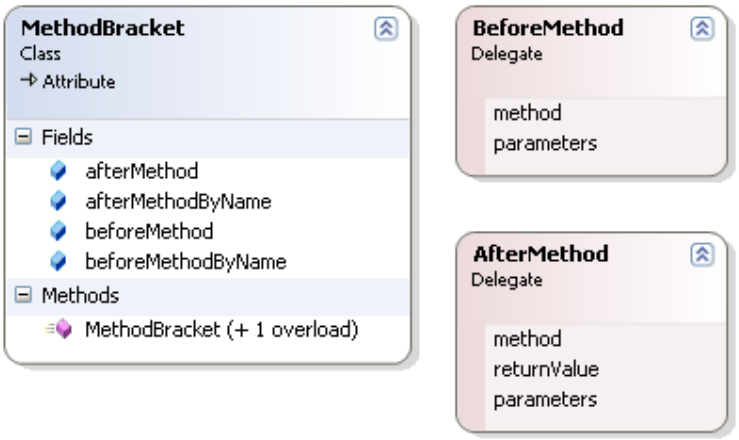

Figure 21. Class diagram for the Bracket concern in the Hyper/Net attributes class library. The respective source code can be found in Appendix - I.1.1.

The *MethodBracket* attribute defines two methods: the before and after methods. These can be identified using one of the two attribute constructors. The first, and the most adequate one, uses method delegates to identify the before and after methods. The second constructor uses method names instead. Still, during runtime, the methods identified by these strings must match the delegate types. We will see how this is implemented by Hyper/Net in the next subsection, when the Bracket concern is analysed for the console application project.

Using the delegate approach is more adequate because type checking is done during compilation instead of runtime. Nevertheless, because the string approach was easier to implement and with the purpose of prototype implementation simplicity, only the methods passed as strings are taken into account. The next subsection describes how bracket composition is implemented using the string representations.

There are two different delegates, one for the bracket before method – *BeforeMethod* delegate – and another for the after method – *AfterMethod* delegate. The *BeforeMethod* delegate provides before methods with a *MethodBase* object (the *method* field of the BeforeMethod delegate in Figure 21) which contains information about the method being bracketed. *MethodBase* is a native class from the .NET framework and is located in the *System.Reflection* namespace. It is the standard way of providing information about methods and constructors in .NET. This delegate also provides the parameters passed to the bracketed method as a parameter object array (the *parameters* field in Figure 21).

The *AfterMethod* delegate provides the same information as the *BeforeMethod* delegate along with an object containing the return value that would originally be returned by the bracketed method (the *returnValue* field in Figure 21). The return value of this delegate will be the actual return value of the bracketed method.

Language Features – Merge

The Merge concern defines the *MethodMerge* attribute (see Figure 22). It is used to declare merge or override composition between methods with the same signature (method name included).

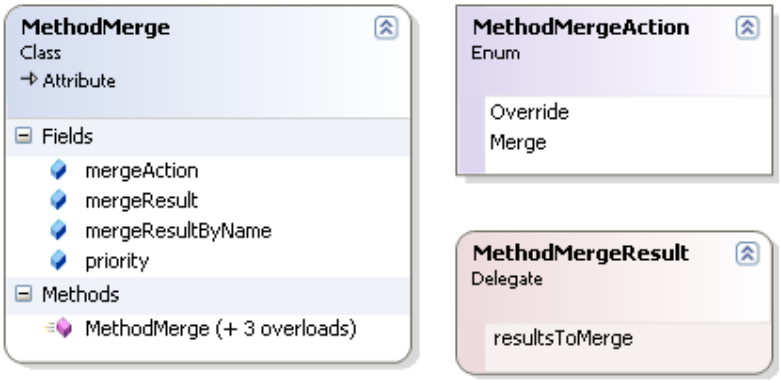

Figure 22. Class diagram for the Merge concern in the Hyper/Net attributes class library. The respective source code can be found in Appendix - I.1.2.

This attribute has four different constructors, each providing a different level of expressivity. The simplest constructor takes only a *MethodMergeAction* argument, initializing the *mergeAction* field. *MethodMergeAction* is an enumerate that distinguishes between override composition and merge composition. In override composition, no other *MethodMerge* fields are used, even if they are initialized. In fact, override composition should be supported using a different attribute. Merge composition requires more details to be provided, has different requirements for usage and the respective validation differs from override composition. Other MDSoC implementations provide separate constructs for merge and override as seen in Chapter 5. This is a design flaw in Hyper/Net that should be corrected as part of future work (see Section 10.3). Even though the remaining fields are not used in override composition when using this first (single argument) constructor, in both override and merge composition, they are initialized with defaults. The priority is set to -1 and there is no *mergeResult* (a *MethodMergeResult* delegate) or *mergeResultByName* string to identify the method used for merging the results of involved methods. This constructor should only be used for override composition or for only one method from a set o matching method in merge composition.

A second attribute constructor also takes as argument the priority level for merged methods. Hyper/Net requires that all merged methods have a different priority level. This imposition exists because of a Hyper/Net implementation detail that can be found in the Merge concern of the Hyper/Net console application project. A .NET *SortedList* object is used to hold data regarding the different methods identified by these attributes. *SortedList* objects do not allow duplicate keys and the priority field is used as a sorting key. More implementation details are provided in the next subsection.

Finally, the two remaining constructors also define a *MethodMergeResult* delegate that identifies a method used to merge the return values of the merged methods. These constructors should be used with methods that return a type. The first constructor of these two uses the typed method delegate directly, storing it in the *mergeResult* field. The *MethodMergeResult* delegate receives the set of return results from merged methods as an object array (*resultsToMerge*) and returns an object with the computed return value. The second constructor of these two uses a method name (a string), storing it in the

mergeResultByName field. The method identified by the string must match the delegate type. Like with the *MethodBracket* attribute, for the purpose of prototype implementation simplicity, only the result merging method passed as a string (*mergeResultByName*) is used.

8.2.2 Command line application

The Hyper/Net command line application is implemented as a C# Console Application project. With the exception of two requirements, which are implemented by the composition attributes class library project, this project implements the requirements that populate the Hyper/Net MDSoC hyperspace.

First off, we analyse the object architecture of this project providing a global perspective of the Hyper/Net internal architecture. Then, the Features and Language Features dimensions perspectives are explored further, as each concern is detailed in terms of implementation.

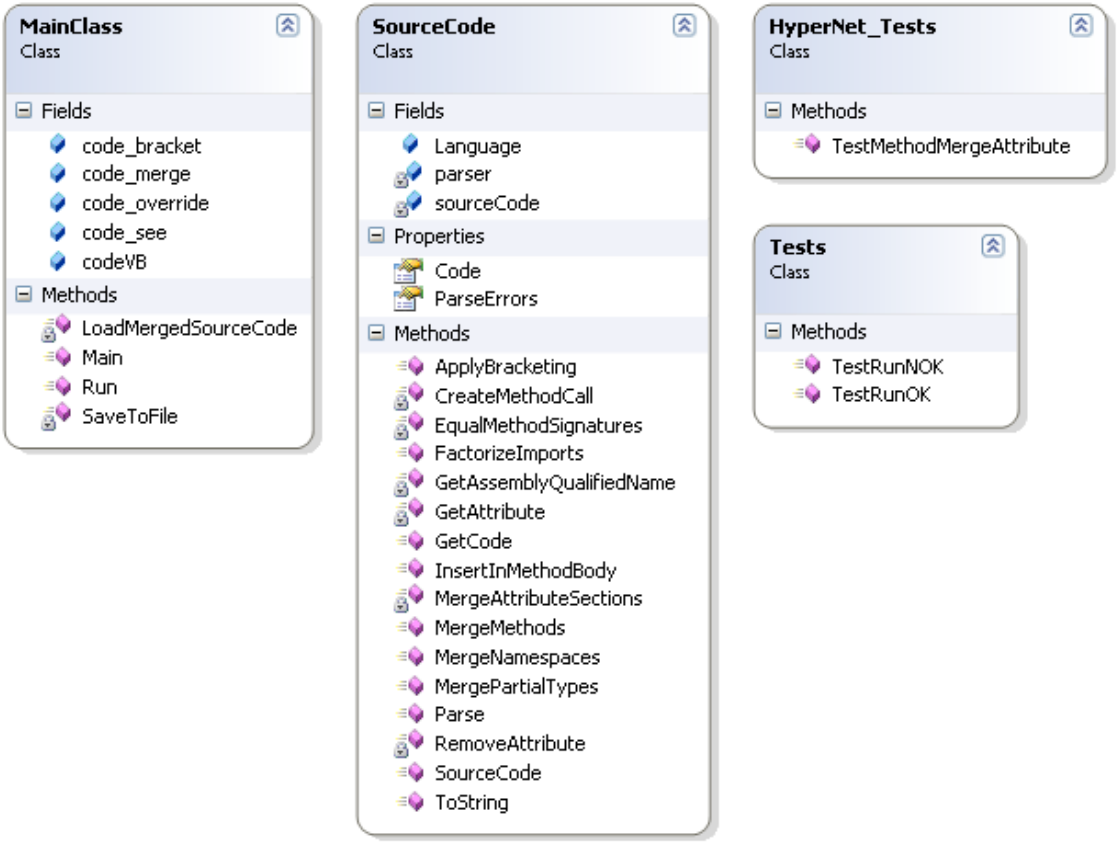

Figure 23. Class diagram for the complete Hyper/Net console application.

The application class diagram in Figure 23 depicts the perspective of the object dimension. The *MainClass* class provides the entry point (*Main* method) that processes command line arguments and takes it from there. Most of the work done is triggered from the *Run* method which will be detailed further on. The *MainClass* also provides implementation for some of Hyper/Net's input and output requirements.

The *SourceCode* class abstracts details like which parser is used or how it is used. Upon construction, it takes as arguments a string of code along with the indication of the language

of that piece of code. That string of code simply concatenates the contents of a set of source code files. The *SourceCode* class then provides methods that prepare this code for parsing, parse it, merge partial types, merge methods, bracket methods and finally provide output code. These methods are invoked, in order, by the *Run* method in the *MainClass*. Each of these methods belongs to a different concern.

Two other classes provide limited testing functionality. This was used and implemented as required during initial development. The *MainClass* fields are only used for these tests.

Features - Flow Control

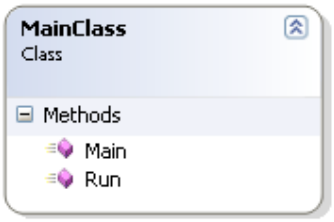

Figure 24. The partial class diagram for the Flow Control concern.
The respective source code can be found in Appendix - I.2.1.

All .NET console applications have as an entry point a *Main* method. In Hyper/Net, this method first checks if the command line arguments are valid. If so, the arguments will identify a project file to be used as input and a target file to write the composed output to. The project file lists the source code files that will be loaded and concatenated into a source code string. To achieve this, the *LoadMergedSourceCode* method, implemented in the Input concern, is invoked. The invocation itself should also be made from inside the Input concern. To achieve this it would be necessary to compose methods, allowing the *Main* method to be decomposed along the different concerns. Unfortunately that falls outside of the partial types MDSoC approach possibilities. In future versions of Hyper/Net, a prior Hyper/Net version could be used for composing the Hyper/Net project, allowing method decomposition to be done.

The string that concatenates the source code is then passed into another Flow Control method – *Run* – also in the *MainClass*. Using a string to contain the concatenated input source code may not be the most adequate approach. As a justification, please recall the prototype nature of Hyper/Net.

The *Run* method invokes the different processing steps in order. The processing steps themselves are factored into other concerns in this (features) and the Language Features dimension. The same decomposition considerations presented for the invocation of the *LoadMergedSourceCode* method also apply here.

The *Run* method receives as arguments the source code to be processed (*code*), an identifier of its language (*sourceLang*), identification of the language in which to write the compose code (*outputLang*) and a path for an output file to store it (*outputFile*). First, the method initializes a *SourceCode* object with the source code. During *SourceCode* initialization, the source code language is also set. Then, the method prepares the source code for parsing by factorizing import/using directives, using the *SourceCode* method *FactorizeImports* from the Parse Preparations concern. Then, it tries to parse the source code. If this fails, the parse error

obtained from the *SourceCode* object is displayed and the source code text being parsed is saved for analysis (using the *SaveToFile* method from the Output concern). If parsing is successful, the remaining steps can be started. All of them use the AST instead of the code in textual form.

In .NET projects, the same namespace is usually scattered throughout different files. This means that there will be several nodes for the same namespace in the AST. So, first, all of the code units that belong to a particular namespace are brought together under a single namespace node in the AST. This is done by the *MergeNamespaces* method from the Namespace Composition concern. Then, inside each namespace, the partial classes with the same name can be composed into a single class. This is done by invoking the *MergePartialTypes* method from the Partial Type Composition concern. Finally, Hyper/Net method composition directives can be processed in the AST. To process merge and override directives, the *MergeMethods* method is invoked from the Language Features dimension, Merge concern. Then, to process bracketing directives, the *ApplyBracketing* method is invoked from the Language Features dimension, Bracket concern. Finally, the output language is used to generate the composed code output in the desired language. This is done by invoking the *GetCode* method from the Parsing concern. The *SaveToFile* method, from the Output concern, is used to write the generated code to the target file.

Unless stated, unrecoverable errors detected inside the invoked methods are communicated through exceptions for which the *Main* method provides a single exception handling point. This exception handling point simply prints the error and returns from the application with an error code of -1.

Features - Input

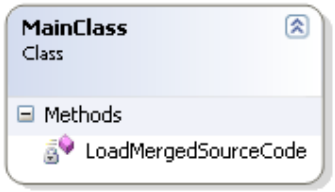

Figure 25. The partial class diagram for the Input concern.
The respective source code can be found in Appendix - I.2.2.

This concern contains a single method of the *MainClass*: *LoadMergedSourceCode*. It receives as input the path to an MSBuild XML project file. It opens that project file using the .NET *System.IO.File* class *Open* method. Then, it initializes a .NET *XmlReader* object that is used to search for *EmbeddedResource* XML elements. These identify the files in the project that were developed using the MDSoC approach and will require composition. For each *EmbeddedResource* element, the *include* attribute is read to get the relative path for the file. Each of these files is then opened (also using *System.IO.File*), read with a *StreamReader* object and concatenated to a string variable. At the end of the process, this string variable holds the concatenation of all code in the project that needs to be composed by Hyper/Net. Finally, the concatenated code string variable is written to a file that can be used to help debugging. The *SaveToFile* helper method, from the Output concern, is used for this purpose.

Features - Kernel

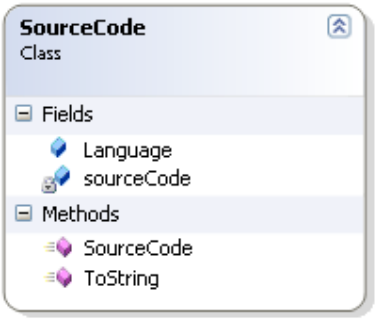

Figure 26. The partial class diagram for the Kernel concern.
The respective source code can be found in Appendix - I.2.3.

This concern provides a constructor for the *SourceCode* object. This constructor takes two arguments. The first is a string containing the source code. The second a *SupportedLanguage* enumeration value which identifies the source code language according to NRefactory classification. Hyper/Net is limited to processing the languages supported by NRefactory. If the parser was replaced, this concern would have to be changed. Instead of using the *SupportedLanguage* as provided by NRefactory, the Parsing concern should hide this enumeration behind one of its own. This way, if the parser was changed, other concerns, like the Kernel concern, would not be affected.

The constructor initializes the respective local variables. One of them, the language of the source code, is made publicly available for reading. The source code string is available indirectly through the *ToString* method, which is also implemented in this concern.

Features – Output

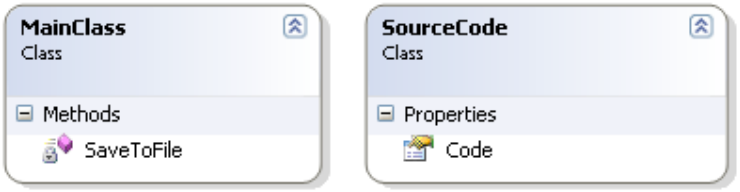

Figure 27. The partial class diagram for the Output concern.
The respective source code can be found in Appendix - I.2.4.

This concern adds an output helper method to the *MainClass*: *SaveToFile*. *SaveToFile* takes as arguments a filename and a string (eventually containing code in text format). It simply writes the string to the file using two objects from .NET *System.IO*: *File* and *StreamWriter*. This method was created to avoid repeating this task in the places where it is used in the rest of the code.

As for the *SourceCode* object, this concern provides a read-only *Code* property. In case the code has not been parsed yet, this property returns the source code string directly. Otherwise, it uses the *GetCode* method, from the Parse concern, to generate a string version of the current

parser AST, in the language of the original source code. Functionality from both the Kernel and Parse concerns is involved in this process, making this concern depend on them. Nevertheless, these can be replaced with equivalent concerns that offer the same functionality.

Features – Parse Preparations

Figure 28. The partial class diagram for the Parse Preparations concern.
The respective source code can be found in Appendix - I.2.5.

This concern adds a *FactorizeImports* method to the *SourceCode* object. This method takes the objects' source code in text form and moves all using/import directives to the beginning of the string. This is necessary to be able to parse the source code.

The implementation is straightforward. The *sourceCode* string variable, of the *SourceCode* object in the Kernel concern, is searched for statements beginning with "using", or "Imports", in case the source code language is VB.NET. Each time such a statement is found, it is added into a hashtable, unless that statement already exists there. It is also removed from the source code. Afterwards, the directives in the hashtable are concatenated at the beginning of the objects' source code string. By manipulating the *sourceCode* variable from the Kernel concern, this concern depends on it.

Features - Parsing

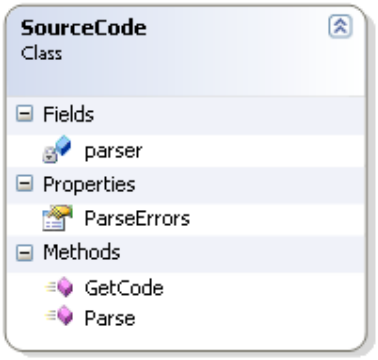

Figure 29. The partial class diagram for the Parsing concern.
The respective source code can be found in Appendix - I.2.6.

Parsing is provided by a *Parse* method in the *SourceCode* class. This method obtains an NRefactory *Parser* object, for the source code language, using the NRefactory *ParserFactory*. The source code string is also passed to the *Parser* object when it is obtained from

ParserFactory. Then, the code is parsed (*Parser.Parse* method). The AST that results from parsing can be accessed through the *CompilationUnit* property of the *Parser*. In case there are errors, the method fails by returning false. These errors can be accessed in string format through the *SourceCode ParseErrors* property. This property obtains the errors in string format from the parser's *Errors* object.

This concern also implements the *GetCode* method, which uses the NRefactory parser to generate a string version of the current parser AST, in the desired language. It does so by creating a language specific NRefactory visitor object, either *CSharpOutputVisitor* or *VBNetOutputVisitor*. This object is then passed to an *AcceptVisitor* method of the *Parser CompilationUnit* property, which holds the AST. Here we are using a visitor design pattern implemented by NRefactory. The *AcceptVisitor* method returns the output code as a string which is then returned by the *GetCode* method. This concern depends on the Kernel concern and on the NRefactory library.

Features – Namespace Composition

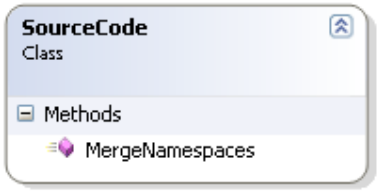

Figure 30. The partial class diagram for the Namespace Composition concern.
The respective source code can be found in Appendix - I.2.7.

Namespace composition is done on the AST that results from source code parsing. This process is implemented in the *SourceCode* class *MergeNamespaces* method. The method iterates through the AST in search of *NamespaceDeclaration* nodes. These nodes hold namespace declarations and, under them, part of the set of code units belonging to that namespace. Each *NamespaceDeclaration* node that is found is added to a hashtable, but only if that namespace does not exist there yet. In case it does exist, all of the *NamespaceDeclaration* node child nodes are copied to the *NamespaceDeclaration* node that was already in the hashtable. The duplicate *NamespaceDeclaration* node is added to a list for removal from the AST, which is done at the end of the method. After this process there will only be a single namespace declaration node for each different namespace and it will contain all of the units in that namespace, independently of the source code file they originally came from. By using the AST, this concern depends on the Parse concern.

Features – Partial Type Composition

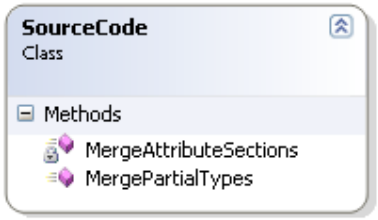

Figure 31. The partial class diagram for the Partial Type Composition concern.
The respective source code can be found in Appendix - I.2.8.

Partial type composition is achieved by the *MergePartialTypes* public method in the *SourceCode* class. It iterates the namespace declaration nodes in the AST. Inside each namespace declaration node, it searches for type declarations (*TypeDeclaration* nodes) that have a partial type modifier (*Modifier.Partial*). This finds partial types inside the same namespace. To match partial types of the same type, the method uses a new hashtable for each namespace that is iterated through. Each partial type found in the namespace is searched for in this hashtable. If it is not found in the hashtable the node is added to it, but, before, its partial modifier is removed. If there already is an entry in the hashtable for the partial type name, all of the found partial type child nodes are added to the matched type in the hashtable.

Different partials classes of the same class type can only extend the same base type, but not all have to. Partial interfaces of the same interface type can extend any interface types. Furthermore, partial classes of the same class type can implement different interfaces. The equivalent whole type has to extend and implement all such types in its partials. To achieve this, each reference in the matched partial types is searched for in the equivalent type in the hashtable. If the reference is not already present, it is added.

Non-repeated type attributes must also be copied from the matched partial types to the equivalent type. Due to its complexity, this is done in a local *MergeAttributeSections* method. A third similar requirement exists for type modifiers. Non-repeated type modifiers must be added from the matched partial types to the equivalent type in the hashtable. This is achieved in a very simple way. As already stated, the partial modifier is removed from all matched partial types. NRefactory represents modifiers as an enumeration. This way, the remaining partial type modifiers are added to the equivalent type in the hashtable using the OR assignment operator (|=) applied to its modifiers and the partial type ones. This results in any non existing modifiers being added to the type in the hashtable.

Finally, all matched partial type nodes that are found are added to a list, except for new ones, which are added to the hasthtable. All members of this list are removed from the AST at the end of the process.

The *MergeAttributeSections* method takes two attribute section lists as arguments. The purpose is to add attributes from the first argument to the second one, without introducing duplicates. In .NET, each attribute can be defined in its own section. But several different attributes can also be declared in the same attribute section. Because of this, the first attribute section list has to be iterated in its sections and in the attributes inside the sections. Each attribute is then searched for in the other attribute section list. If it does not exist there, then it is added to a return attribute section list. Finally, all of the attribute sections in the second argument are also added to the return list.

This concern focuses on additive merging at class level. After merging partial types, there can be repeated method signatures inside classes. This is an issue for the Merge concern in the Language Features dimension, which is analysed further on.

Language Features – Bracket

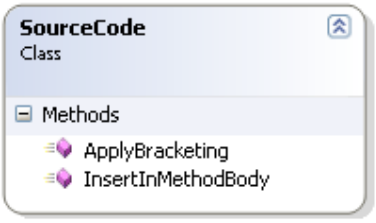

Figure 32. The partial class diagram for the Bracket concern.
The respective source code can be found in Appendix - I.2.9.

This concern introduces the *ApplyBracketing* method which transforms the AST by executing bracket composition. It iterates the AST hierarchically to find method declarations with the *MethodBracket* attribute. Each method is searched for a *MethodBracket* attribute using the *GetAttribute* method. This method currently belongs to the Merge concern. The Language Features dimension may not provide the best location for such shared helper methods. Possibly a new dimension should be introduced as these methods do not fit the Features dimension either. But this is an issue for future refactoring.

When a method that has a *MethodBracket* attribute is found, the attribute is removed from it. This is done using another helper method, *RemoveAttribute*, also contained in the Merge concern. Then, a statement that places the method meta-information into a variable is injected at the beginning of the method. This is done by creating a *LocalVariableDeclaration* statement, using AST classes that represent fields and method invocations. This statement populates a *MethodBase* object from the .NET *System.Reflection* namespace using the static *GetCurrentMethod* method, found in the *MethodBase* class. A local method (*InsertInMethodBody*) was created to help introduce these statements at specific points of the method body. In this case, the statement must be inserted before the remaining method body (position zero). The meta-information about the method will be passed to before and after methods, when they are invoked.

At this point, if the *MethodBracket* attribute defines a non null *beforeMethodByName* variable, a method invocation statement is created and introduced before the method body original code. Again, the invocation statement is constructed using AST classes. The *beforeMethodByName* variable is used to identify the method that is invoked. This method receives two arguments: the base method meta-information variable and a parameter array containing the parameters received by the base method. Also using the *InsertInMethodBody* method, this invocation statement is introduced in the second position of the method body, after the method's meta-information variable is populated.

Finally, if the *MethodBracket* attribute defines a non null *afterMethodByName* variable, a method invocation statement is created. Then, the base method body is searched for return statements. Each return statement is replaced by an invocation of the after method. This invocation receives the same arguments as the before method and, additionally, the original return statement expression. In case no return statements are found during this process, then, an invocation of the after method is created with the same arguments as the before call and a null return value argument. It is added at the end of the base method body.

Adding a statement to the end of a method body using NRefactory is trivial. The statement is simply added as a child node of the *method.Body BlockStatement* object. Replacing a

statement node (like return statement nodes) is also trivial: the *Expression* field of the statement is simply replaced. Still, the method body is represented as a *BlockStatement* object. This object does not allow adding statements except at the end of the block (method body). To add statements to a specific position (like the beginning of the method body), the *InsertInMethodBody* method had to be introduced. This method iterates through the statements in the method body, from the end to the position where the new statement is to be added. Each statement iterated through is removed into a .NET *List*. The new statement is added to the method body when the adequate position is reached by this iteration. At this point, all subsequent method body statements have been removed. The new statement can simply be added like a last statement of the method body. Afterwards, the statements kept in the list are added back at the end of the method body. This is done in the reverse order of which the statements were introduced into the list.

It is easier to cope with necessary changes to attribute definitions by having this concern exist, in a decomposed form, in the console application project. If bracket attributes have to be extended for further expressivity, the only code in the console application project that is affected will be located in the Bracket concern. Furthermore, a version of Hyper/Net without bracketing features could be built by removing the directory for this concern from the project. The invocation of the *ApplyBracketing* method from the *Run* method in the Flow Control concern also has to be removed. If the current Hyper/Net version was used to decompose Hyper/Net source code methods, namely the *Run* method in the Flow Control concern, the invocation of the *ApplyBracketing* method could also be decomposed into this Bracket concern.

Language Features – Merge

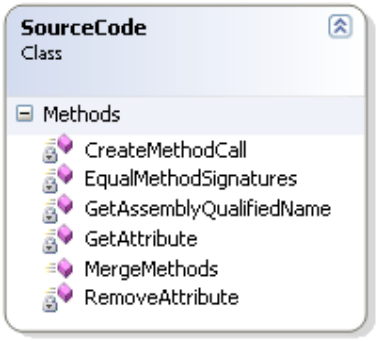

Figure 33. The partial class diagram for the Merge concern.
The respective source code can be found in Appendix - I.2.10.

This concerns offers merge and override method composition through the *MergeMethod* public method. It also contains a set of necessary support methods which are private. Unlike the bracket implementation, *MergeMethod* does not directly search for particular attributes. Yet, it uses a similar hierarchical iteration in namespaces, their types and, then, each type's methods. For each method found, the containing class is searched for all methods that have the same signature. The comparison between each pair of methods is done using the *EqualMethodSignatures* method. If another method with the signature of the current method is found, any *MethodMerge* attribute applied to it is obtained (using the local *GetAttribute* method) and its *MethodMergeAction* is checked. This allows determining the type of composition defined: override or merge. When an override *MethodMergeAction* is found, if

another such attribute had already been found for the method signature being searched for, the *MergeMethod* method throws an exception, stating that only one override composition attribute is allowed. Similarly, if override attributes coexist with merge attributes, for the same set of matching methods, an exception is thrown, because override composition is incompatible with merge composition.

When all the methods inside the class have been compared with the current method, it is known how many matching methods exist and how these should be composed. The simplest case is when there is a single method and it has a merge attribute applied to it. This case is dealt with by removing the attribute from the current method and continuing the search with the following methods in the class. Attributes are removed using the local helper method: *RemoveAttribute*.

If more than one method was found with the same signature and none had composition attributes, then an exception is thrown, stating there must be at least one *MergeMethod* definition for repeated methods.

If an override attribute was found, the matching method declarations in the same class are iterated again and all, except the one with the override attribute, are added to a list of methods to be removed. At the end of the iteration inside the class, this list will be iterated through and all methods in it are removed from the class.

Otherwise, if the attributes define a merge composition with more than one method involved, a new method declaration is constructed. This declaration has the signature of the matching methods, including the name. Its body, a *BlockStatement* object, will then be populated. To help in this process, two lists are kept, while the current class is iterated through once again. One of these lists will hold the method invocations for the methods that are being merged, in the order defined by their priorities. The other, will hold the return variables where the results of each of these method invocations will be kept. First, the current class is iterated through once again, to locate the methods being merged. Each method is renamed, by concatenating an increasing integer variable to the method name. It is also made private. A method call statement for this renamed method is created using the local method *CreateMethodCall*. Next, the priority of the method and a merge results method are obtained, if present in the methods' *MethodMerge* attribute. In case the method invocation returns a value, a variable to contain the result is also declared and initialized with the invocation of the renamed method. The uniqueness of its name is guaranteed like with the renamed method name, using the same integer variable. In this case, the result variable is added to the list of return variables, using the inverse order of the priority declared in the attribute. It is also added to the invocations ordered list, also using the inverse of the attribute priority. In case the methods have a void return, only the renamed method invocation is added to the invocations list. The *MethodMerge* attributes of the renamed methods are removed, again using the *RemoveAttribute* method. Finally, the attributes of the renamed method are moved (copied and removed) to the new merged method declaration.

When all the matching methods in the class have been iterated through, the renamed method calls are introduced into the merged method body, according to their order of priority. If there is a result merging method, a new variable is declared and initialized with a call to the result merging method. This call is done with a list of arguments obtained from the list of return variables. This variable declaration is added at the end of the merged method body, along with a return statement, returning this variable.

Finally, the merged method is added to a list of methods that are to be added to the class. At the end of class iteration, this list is iterated through and any contained methods are added to the class.

The *GetAttribute* method abstracts the complex process of getting the first attribute, in a particular method, which has a specific attribute type. Different attribute sections can exist; these have to be iterated through, as well as the contained attributes themselves. When an attribute with the desired type is found, it has to be processed for arguments. Each argument has to be converted from an AST expression form into the adequate .NET type and have its value extracted. This is done by analysing the different possible kinds of attribute argument expressions, adequately extracting their type and value, processing the value (for instance, parsing an integer representation) and adding it to a list. Finally, this list is used to initialize an object of the attribute type.

Fortunately, removing an attribute is simpler. Still this task was abstracted into the *RemoveAttribute* method. Like *GetAttribute*, it also iterates attribute sections and their attributes. Any attributes that match the desired type are added to a list for removal. This is done after the each attribute section is iterated through. If the attribute section itself is empty after the attribute type is removed, it is also added to a list for removal at the end of the process.

The *EqualMethodSignatures* method first checks if the two method declaration objects (*MethodDeclaration*) that are passed as arguments return the same type. If not, their signatures are not equal. Then, it compares the number of arguments of each method. If they are equal, it compares the argument types one by one. If everything matches, the methods have the same signature. The method names should also match, but this check is done outside this method. This allows using it for a more generic comparison of method signatures, without comparing method names.

The AST structure for calling a method can be obtained using the method declaration itself. Still, this structure is not straightforward. The *CreateMethodCall* method was introduced to abstract this process. First, a *FieldReferenceExpression* object is created from the method name. It is used to create an *InvocationExpression* object to which the method parameters are added, one by one. This assumes that the method parameter names are available at the calling scope. The *InvocationExpression* object is then returned.

8.2.3 Testing Hyper/Net

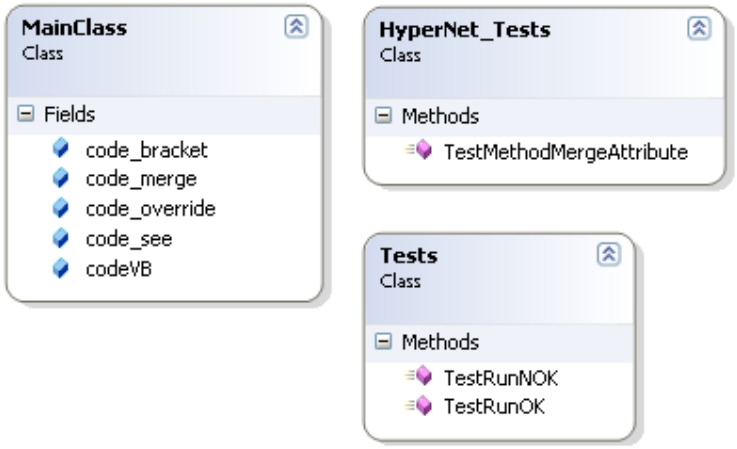

Figure 34. The partial class diagram for the Tests concern.
The respective source code is not available because this is a minimal test implementation. There is a lot of sample code involved in the tests so it would take up much space without providing enough benefits.

The code that provides testing functionality for Hyper/Net could be seen as defining its own dimension. Due to its size, it is only made up of a single concern. There has not been much emphasis on testing Hyper/Net functionality thoroughly yet. There are only a few detailed tests, like the *TestMethodMergeAttribute,* that validates basic features of the *MethodMerge* attribute with an override *MethodMergeAction*. The *MainClass* also contains several blocks of code focusing different situations with bracket, merge and override composition. It also contains a VB.NET code example and an example that is invalid C# code. All of these can be processed using the two *Tests* class methods.

This concern is a good example of a concern where the object dimension is locally representative. As for the rest, only another concern, the Output concern, contains more than one class. The other concerns are not so balanced because Hyper/Net is mostly broken down into the *MainClass* and the *SourceCode* classes. *MainClass* captures elements from the Flow Control and I/O concerns. *SourceCode* captures elements from the Language Features concerns and Features concerns that used to prepare the code to be processed by the language features. Some code refactoring, that should be done to enhance comprehensibility and Hyper/Net evolution, is expected to develop the object dimension further and, possibly, end-up with a more dispersed object dimension, in relation to the other dimensions of concern.

8.3 Conclusions

One of the most adequate ways of looking at Hyper/Net is from the perspective of the Time dimension. As with normal command line applications, Hyper/Net processing starts and ends with I/O oriented tasks. It processes entire .NET projects, but only a single source code file is generated. The reason for this is that the parser used by Hyper/Net can only process a single block of source code. This requires that the source code from the files in the project is concatenated into a single string. Preparing this string for parsing and parsing it are the two processing steps that follow the initial input steps. Before the actual Hyper/Net attribute composition takes place, two other native .NET composition steps are taken, composing namespaces and partial types. Merge and override composition is processed first, in a single step. It is followed by bracket composition which leaves the code ready for output. All of the composition steps work with the AST version of the source code.

While the Time dimension is a virtual one, there are four other dimensions in Hyper/Net that are physical. A project dimension divides Hyper/Net into a public class library, which should be referenced from projects that use Hypert/Net, and the actual source code processor, which is a console application. There is also a Features dimension where a Flow Control concern operates the different processing stages. It does this by invoking other features, which are also concerns in the Features dimension, and Hyper/Net specific language features. These specific language features implement and provide declarations for Hyper/Net attribute composition and belong to their own dimension, the Language Features dimension. Finally, there is a Test dimension with simple ad-hoc tests. Curiously, the Features and Language Features dimensions, together, almost implement the (virtual) Time dimension.

By adopting the MDSoC organization of Hyper/Net for this chapter, it was fairly simple to address each requirement implemented by Hyper/Net down to the full detail of the

implementation (code artefact). Each concern is focused at a time, providing a simple yet complete and detailed description.

Chapter 9

Comparing MDSoC implementations

The topics addressed in the previous chapters are inter-related, in particular the MDSoC implementations presented in Chapter 5 and Chapter 6. But these were not compared, except for some particularities. This chapter provides such a comparison between MDSoC implementations. This comparison is done according to groups of criteria, similar to those used to analyse each implementation. The evaluation according to these criteria is summarized in a table. Then, each group of related comparison criteria is analysed in more detail in its own section.

9.1 Comparison criteria

The criteria for this comparison were chosen according to the criteria by which each MDSoC implementation was analysed in Chapter 5 and Chapter 6 and can be grouped as follows:

- Context – The first set of comparison criteria addresses the context in which the implementations can be used. These range from the artefacts and formalisms (programming languages) that can be used, to the context of these solutions in terms of the standard compilation process and whether they require using any additional software.

- Hyperslices – Compares the hyperslice implementations that can be used, the smallest composable units and whether declarative completeness can be achieved.

- Hypermodules – Focuses on the implementation of composition provided by each solution. The individual criteria are: the number of coexisting hypermodules that are possible, whether the composition definition is mixed with the code, which composition strategies and exception relationships are usable and, finally, which composition functions are supported.

- Reuse – Addresses hyperslice and hypermodule reuse limitations.

- Usage – Compares how the implementations can be used for different tasks and scenarios. The usage scenarios that are addressed are: usage from scratch, to introduce new features to existing code, to decompose existing code, to mix-and-match concerns and usage without source code.

- Other limitations – Address additional limitations that could not be captured with the other criteria.

9.2 Comparison summary

Table 1 summarizes the comparison of MDSoC implementations in terms of the most relevant points. Each is addressed in more detail afterwards. In this table, the most interesting results for each point of comparison are highlighted in bold.

		Hyper/J	HyperC#	Partial Types	Hyper/Net
Context	**Artefacts**	Code	Code	Code	Code
	Formalisms	Java	C#	**Any .NET 2.0 (or above) language**	**C# and VB.NET**
	Compilation context	Intermediate language weaver	Source code transformation	Source code transformation	Source code transformation
	Introduces additional software	Yes	Yes	**No**	Yes
Hyperslices	**Hyperslice implementation model**	**Physical + Virtual**	Physical + Limited virtual decomposition	Physical	Physical
	Composable primitive units	**Type members (methods, variables, etc.)**	Methods	Partial types	Methods
	Declarative completeness	**Possible**	Not supported for methods	Possible for types (classes, interfaces)	Possible for types and methods
Hypermodules	**Supported hypermodules**	**∞**	∞, each limited to a single class output	1 per .NET project	1 per .NET project
	Composition definition separate from code	✓	✓	✗	✗
	Composition strategies	**1 per hypermodule**	1 per hypermodule[27]	Static: composition defined by partials	Static: composition defined by partials
	Composition relationships	**Scope: all units but packages**	Limited to 1 bracket and 1 equate[28] (for methods)	None	Scope: only for methods
	Supported composition functions	Merge, Override, Bracket	Merge, Override, Bracket	-	Merge, Override, Bracket
Reuse	**Hyperslice reuse**	✓	✓	Need to anticipate composition in reuse scenario	Need to anticipate composition in reuse scenario
	Hypermodule reuse	✓	Limited by the single class output	✗	✗
Usage	**Used to decompose existing code**	✓	✓[29]	✓	✓
	Used from scratch	✓	✓	✓	✓
	Used to introduce new features	✓	✓[29]	✓	✓
	Used without source code	✓	✗	✗	✗
	Mix-and-match	✓	✓	✓	✓
	Other limitations	Significant implementation limitations that invalidate certain features (see Subsection 5.1.4)	Developers need to program in an extremely limited GUI	-	-

Table 1. Comparison chart of MDSoC implementations.

[27] But only generates a single output class.
[28] Equate applies the composition strategy composition function to any pair of non matching methods.
[29] But the code needs to be input manually in the HyperC# GUI.

9.3 Context

All of the four MDSoC implementations presented herein are limited to the code artefact. Two of them (Hyper/J and HyperC#) are limited to a single language while the remaining two (.NET partial types and Hyper/Net) support more than one language, but not simultaneously.

Only the first of these MDSoC implementations, Hyper/J, operates after source code is compiled (into Java bytecode). The remaining implementations work previously in the compilation process, as source code pre-processors. These implementations output composed code that afterwards is compiled using standard compilers.

All MDSoC implementations analysed but one need to introduce additional software that is specifically targeted for MDSoC composition. .NET partial types are a native feature of .NET 2.0 languages and support a basic MDSoC model that requires no specific software, other than a standard .NET compiler. This fact has the advantage that there is already a large user base that can benefit from this approach in their existing development environments. The other implementations require installing software and, usually, setting up the development environment for MDSoC.

9.4 Hyperslices

Another comparison element is the way MDSoC hyperslices can be implemented in each approach. Hyper/J is the only approach that supports near limitless virtual decompositions. HyperC#'s support for virtual decomposition is limited to decomposing methods virtually. This is further limited by a physical implementation for the virtual method decompositions. Hyper/Net does not support virtual decomposition at all.

In MDSoC, virtual decomposition is important when units are indecomposable. But, when units are actually decomposable, the physical decomposition model should be used instead. Fortunately, the physical decomposition model is supported by all of the analysed approaches. Each approach supports the physical hyperslice model using different structural elements. Hyper/J proposes "Hyperslice Packages", where each hyperslice is implemented by a particular package (see Subsection 5.1.1). Hyper/Net and the partial types approach use directories in a similar fashion (see Section 6.1). Finally, HyperC# uses a GUI to represent hyperslices and their contents, but behind the scenes implements hyperslices physically in different classes that are prefixed with the hyperslice name (see Subsection 5.2.1).

The physical hyperslice implementation model allows developers to manipulate the MDSoC hyperspace structure and contents directly. It provides something tangible that developers can see and work with. Also, by structuring hyperspaces using physical entities that most developers are already acquainted with, namely packages and directories, hyperspaces become more familiar. These advantages of the physical hyperslice model make it the model of choice whenever it is possible to physically decompose units. Units should only be matched to more than one hyperslice, using virtual decomposition, when they cannot be physically decomposed. Virtual dimensions will provide alternative views of a hyperspace. For the benefit of developers, virtual dimensions can become almost as usable as physical dimensions by allowing their direct manipulation, for instance, using IDE visualization and editing plug-ins. Yet, neither of the implementations that somehow support virtual decomposition (Hyper/J and HyperC#) offers such a feature.

Another defining aspect of MDSoC implementations is the granularity down to which it is possible to decompose units that latter can be composed. As defined in Section 3.2, the smallest units resulting from decomposition are primitive units. Different MDSoC implementations support composing primitive units at different levels of granularity. Supporting lower levels of granularity provides more power in decomposition and composition. Methods are primitive units in Hyper/J, HyperC# and Hyper/Net. The .NET partial types approach only supports primitive units at a higher level of granularity: partial types, which are slightly below the granularity of classes. While the only primitive units supported by HyperC# and Hyper/Net are methods, Hyper/J supports composing other kinds of primitive units, like variables and other class members. Simultaneously, Hyper/J supports composing units at higher levels of granularity. This is crucial in terms of the expressive power of MDSoC implementations. The other approaches also support higher level compositions, at class level, but only in a static fashion. HyperC# composes all input classes into a single class, while .NET partial types and Hyper/Net compose matching partial types into a single type.

As MDSoC hyperslices can reference units that are outside them, some consider that these hyperslices should be declaratively complete. Declarative completeness is fully possible with Hyper/J. As for the remaining implementations, it is only possible for particular unit types, like types and, in the case of Hyper/Net, also for methods. We consider these limitations to be unimportant. Our view on the need for declarative completeness has already been presented at the end of Subsection 6.1.1. Declarative complete hyperslices that are not composed with other hyperslices, offering the required units, will only yield errors during runtime, when these units are used. Without declarative completeness these errors will be detected earlier on, during compilation, which is always better for the developer.

9.5 Hypermodules

As for composition, all the implementations allow the creation of as many hypermodules as required for a given hyperspace. But only Hyper/J has no particular restrictions on these hypermodules. HyperC# only outputs one class for each hypermodule. This does not allow using hypermodules to directly create most of the outputs desired (for instance, a class library), which usually need to include more than one class. There are workarounds for this issue. Namely, to bring each class that is output by a different hypermodule under a single project that is then compiled. Yet, this is not a natural way to program with MDSoC because it forces developers to think in terms of the classes that are output, one by one. It is clearly forcing developers to think in terms of the object dimension. As for the remaining implementations, both Hyper/Net and the partial types approach need to use one project to implement one hypermodule. Apparently, this could be almost as limited as having hyperspaces made up of a single project and having only one hypermodule defined for it. In particular, a hypermodule that always includes the entire hyperspace. But, in fact, with both approaches, it is possible to have hyperspaces that span several different projects. In these hyperspaces, hypermodules can compose units from different projects by being defined through a new project which includes the desired units using one of the linking approaches discussed in Subsection 6.1.3. This solution is almost transparent for the programmer and provides as much versatility as Hyper/J's multiple hypermodules.

Hyper/J and HyperC# allow defining a composition strategy that is applied to each hypermodule. In Hyper/Net and the .NET partial types approach, the composition strategy is statically defined by partial type composition. With their static composition strategy, these

approaches have a more limited composition expressiveness that cannot be overcome with composition relationships.

When it comes to composition function support, all implementations, except for .NET partial types, offer the same composition functions: merge, override and bracket. .NET partial types implement a single, merge-like, composition function for partial types. It is applied as part of the .NET partial types composition strategy and cannot be applied in an ad-hoc fashion (as composition relationships). In Hyper/J, all composition functions can be used in an ad-hoc fashion in composition relationships, involving all units but packages. As for Hyper/Net, it only supports composition relationships involving methods. HyperC# is also limited to the ad-hoc composition of methods. It further limits this composition by allowing only the usage of one bracket composition function and one equate composition relationship per hypermodule. Recall, from Subsection 5.2.2, that the equate composition relationship will apply the composition function defined by the composition strategy to a given pair of non-matching methods.

The composition information in a hypermodule can be seen as metadata. It can coexist with the code, sometimes being part of it, or, it can exist separately, referencing the code as required. The first option was taken with the partial types approach and Hyper/Net, where composition information coexists with and is part of the code. On the contrary, in Hyper/J and HyperC#, it exists separately from the code, in specific files. The main advantage of having composition information separate from code is the ability to reuse either (code or composition) whenever it is appropriate.

9.6 Reuse

Hyper/Net and the partial types approach allow very limited reuse because they mix composition information with the code itself. When composition information coexists with code, it might be easier to understand the units that are involved in compositions, by having that information next to the code it affects. This is the only advantage of this coexistence. When composition information is separate from code, the same advantage can be provided by specific tools that read composition information and represent it next to the code it affects. Still, none of the analysed implementations offers such tools at this point.

In terms of hyperslice and hypermodule reuse, Hyper/J provides the best support, allowing hyperslice and hypermodule reuse without particular limitations. HyperC# follows, with the only limitations imposed on hypermodule reuse by its single class output. Hyper/Net and .NET partial types provide very limited reuse features. They do not allow hypermodule reuse due to relying on partial type composition, which can only take place inside the same project and cannot involve types that are not partial. This excludes the types that are output by hypermodules. As for hyperslice reuse, both implementations are limited by the need to anticipate the reuse scenarios by introducing partial types that match for all expected reuse scenarios. This is a consequence of mixing composition information with code in both implementations.

9.7 Usage

In terms of usage scenarios, all implementations provide similar support. All can be used from scratch, to decompose existing code, to introduce new features into existing indecomposed code and to mix-and-match concerns. Due to its context in the compilation process, the only implementation that supports composing software without source code is Hyper/J. It can be used to decompose, mix and extend existing applications for which there is no source code. The other implementations lack these features.

9.8 Other Limitations

Hyper/J is implemented with several technical limitations. These disallow some of the features discussed. Still, Hyper/J should support the full announced feature set if these technical issues are overcome. As is, the most relevant effects of these technical limitations are disallowing composition relationships using merge and override composition functions and outputting all composed units as public. As for HyperC#, it is possibly the least programmer friendly approach, due to its need to use a specific GUI to define classes. This GUI is extremely limited and unnatural. If HyperC# used a parser to gather metadata about the code, it would not need the GUI and would provide a much more interesting implementation. As for .NET partial types and Hyper/Net, no limitations other than the ones already mentioned have been identified.

An unfocused element of comparison is support for traceability of compilation errors and debugging. Implementations working with source code are usually more limited in this field as they rely on specific compiler and debugger support for mapping the composed code to the original code. Solutions that work after the decomposed code is compiled can rely on standard compiler and debugger support for the decomposed code, providing a more adequate developer experience. This way, Hyper/J is expected to provide the best support in this regard. As for the other implementations, by being a native approach, using standard .NET compilers, the partial types approach also provides error reporting and supports debugging relative to the decomposed code. Hyper/Net also provides some level of error reporting relative to the decomposed code, in particular, parsing and composition errors.

9.9 Conclusions

Hyper/J has been available for download from IBM since 2000 and the latest version available is from 2003 [HyperJ03]. It can be considered the less limited MDSoC implementation, with rich composition expressiveness (which can be used at diverse granularities), excellent reuse capabilities and being the only one that is able to decompose compiled code. HyperC# is not publicly available and is mostly limited by its particular GUI approach that forces developers out of their IDEs. The biggest limitations in .NET partial types and Hyper/Net are dictated by mixing composition information with code, which is imposed by the usage of partial types as a native composition mechanism. Enhancing Hyper/Net reuse is one of the major issues presented for future work in Section 10.3.

We wrap-up our analysis by highlighting the points where each implementation surpasses the others:

- Hyper/J: composition expressiveness (at diverse granularities); excellent reuse.
- .NET partial types: being native.
- .NET partial types and Hyper/Net: supporting more than one language.

It would be desirable to agglomerate these features in a single MDSoC implementation.

Chapter 10

Conclusions and Future Work

This chapter is organized in four sections. Section 10.1 provides an evaluation of the results achieved as part of our thesis. It also tells of additional knowledge transfer experiences related with Hyper/Net. Section 10.2 presents the history of the development of Hyper/Net and summarizes parallel developments and investigations, not presented in this document.

Section 10.3 presents several topics that should be addressed by future work. Most future work is directed towards Hyper/Net, which at least requires to be leveraged with the features of other implementations, in particular, Hyper/J. A more ambitious perspective sees Hyper/Net as a prototype with which it is possible to gather and evaluate more elements regarding different aspects of MDSoC composition. This allows addressing other issues for future work that fall outside the strict boundaries of the code artefact, under a unified MDSoC perspective.

To end this chapter, Section 10.4 provides a few final remarks.

10.1 Results

Recall, from Section 1.1, that this thesis had three chained goals. The first was to discover or develop a method to use MDSoC while programming with .NET. The next was to implement the classic Expression SEE example using this method. Finally, the last was to validate the results of the Expression SEE example with adequate tests, in particular, ones that could test different flavors of the example obtained by mix-and-match.

With the .NET partial types approach and Hyper/Net, the first goal was achieved. These approaches have reuse limitations, but reuse was not the focus of our goal. There are also some other limitations, focused in Subsection 6.2.3 and in Chapter 9, but these may be overcome in future versions of Hyper/Net as will be proposed for future work. Both approaches allow programming in the existing development platforms and have little impact on existing development processes, apart from allowing the adoption of MDSoC.

As part of our goals, our MDSoC implementation for .NET had to be validated. We followed two different paths to do this. The first was to implement and thoroughly test a classic

MDSoC example. The second was to validate our work with peers from different communities by means of public presentations.

10.1.1 The Expression SEE case study

With Hyper/Net, it was possible to implement the Expression SEE example from MDSoC literature, as documented in Section 7.5. Only the logging feature could not be implemented as easily as documented in MDSoC literature due to the limited Hyper/Net matching mechanism. Caching was another feature proposed in MDSoC literature [Ossher99] but could not be easily implemented in either Hyper/Net or Hyper/J due to the lack of support for around functionality in bracket composition.

Finally, the expression SEE example was tested using unit tests. Composed behavior was adequately tested using the override and merge composition of unit test methods. Our testing approach showed it could cope with the removal and introduction of the concerns in this example, thus, being compatible with mix-and-match.

10.1.2 Public presentations

Apart from this document we were also able to validate the results by both publishing a paper with some results and then performing public presentations to communities from the fields of Aspect-Oriented Software Development (AOSD), Intelligent Transport Systems and Microsoft .NET software development.

The first public presentation of our work was done at an AOSD workshop in Spain [Dias06]. There, we presented an overview of Hyper/Net and demonstrated its usage to extend a basic version of the Toll example presented in Section 7.4. We also did a presentation on the usage of design-patterns and MDSoC for Intelligent Transport Systems software, as part of a workshop promoted by a Portuguese motorway operator [Dias07]. This presentation referenced the advantages of MDSoC in comparison to design-patterns, when used for ITS software, also using the Toll example.

Finally, in March 2007, we were invited to present this work in an hour and a half session at Microsoft's TechDays event in Portugal. This event is directed towards developers and IT professionals working with Microsoft technologies, which are the focus of most sessions. Our session was titled "Aspect-Oriented Programming in .NET" and was included in the development track of the event. It introduced Aspect-Oriented Programming, presented and compared AOP solutions for Microsoft .NET and showed how MDSoC could be used to introduce a symmetric approach to address the same issues as AOP. This session also provided live demonstrations using an AOP tool (AspectDNG) and Hyper/Net. The attendance figures for this session, provided by Microsoft, were unexpected. The room only had 20 seats but it was able to accommodate the 58 attendees. Out of these, 40 filled in an evaluation form, rating the session at 6.36 on a score from 1 to 9.

Papers and presentations for all these sessions are publicly available at the author's homepage: http://ptsoft.net/tdd/. Some materials are only available in Portuguese.

10.2 The history of Hyper/Net

Hyper/Net started out as an implementation project to support the writing of a paper for a course on Advanced Topics in Software Engineering, which is part of the MSc curricula. We started out by using partial types to implement the Expression SEE example in C#. Soon, we felt the need to compose methods so we could fully implement the Expression SEE example, but, partial types do not support this. At that time, there were no MDSoC implementations that supported C#. HyperC# was not publicly available and a paper about it [Hantelmann06] was still to be published. This was when we decided to implement Hyper/Net and use it to compose the code for the finished Expression SEE example. The results of this work were summarized in a paper for that course and presented in class. This paper was selected for submission to a workshop on AOSD, held as part of the JISBD conference in Spain[30]. It was adapted and presented at the workshop on October 2006 [Dias06].

The paper presented in Spain already identified the .NET partial types MDSoC approach and provided an example on how to use Hyper/Net in complement to this approach. Using existing development platform features to natively implement MDSoC was an innovation. This was why we chose to extend these topics for the MSc thesis documented herein. Much of our initial work was carefully reviewed and the conceptual model of our approaches was documented according to the MDSoC model presented in literature. Other MDSoC implementations were also further analysed, namely HyperC#, as [Hantelmann06] was finally published in the mean time.

During the elaboration of our thesis, we also dedicated efforts to contextualize the MDSoC approach in terms of human cognition and in regards to other composition solutions, namely AOSD. These branches of our investigation have already provided interesting results and will certainly be focused in future publications. Including them in the context of our MSc thesis might risk dispersing the focus of our work, so we chose not to address these contextual topics. At this point we have done and have ongoing work on:

- The investigation of the relations between MDSoC and the human cognitive model, as seen by cognitive sciences.
- Completing the formal MDSoC model started by [Ossher99], according to information provided in an informal fashion in [Ossher99] and other MDSoC literature.
- Using the MDSoC formal model to evaluate MDSoC implementations, including Hyper/Net.
- Using the formal model to compare MDSoC and AOSD.

10.3 Future work

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

[30] The homepage of the AOSD workshop at JISBD 2006, DSOA'06, can be found at http://www.dsi.uclm.es/personal/ElenaNavarro/DSOA06/.

Hyper/Net still has a long way to go in terms of composition expressibility. Extensions to the present Hyper/Net composition constructs are required, some more trivial than others. There is also a clear need to find a composition representation that supports better hypermodule and hyperslice reuse. To achieve this, Hyper/Net should be reviewed in terms of implementation. But, above all, there should be a prior analysis and design work around existing and missing functionalities.

10.3.1 Extensions to Hyper/Net composition

Hyper/Net uses the same attribute for expressing merge and override composition. This is an unnecessary confusion resulting from a design flaw. These should be expressed separately, by two different composition attributes. These two attributes could share common features by extending the same base composition attribute class.

Mix-and-match is not suitably supported by override composition. Removing a concern which has an override composition attribute applied to a particular method will leave the remaining set of matching methods un-composed. To avoid this, it should be possible to introduce multiple override composition attributes and decide on which to enforce, based on the priority of each attribute. This would also benefit the composition of interfaces. Still, Hyper/Net could compose partial interfaces without requiring any composition attributes. Interfaces only provide member declarations and these must be equal in partial interfaces, so they can be matched. Hyper/Net could simply remove repeated declarations while composing partial interfaces. Composition attributes applied to interface elements would simply be ignored. However, this is only a hypothesis. An adequate design decision for interface composition requires further research, for example, in terms of what to do with member declaration metadata, like attributes and XML documentation comments.

Hyper/Net only supports method composition. The composition of constructors, properties, and, eventually, variables should also be considered.

10.3.2 Extending Hyper/Net support for reuse

According to the MDSoC model, hypermodules should be reusable in new compositions (see Section 3.1). By allowing the definition of different hypermodules for the same hyperspace, the MDSoC model also supports hyperslice reuse. Subsection 6.2.3 identifies severe limitations in hyperslice and hypermodule reuse with Hyper/Net. That subsection traces the origins of these limitations to two facts:

- Hyper/Net uses units from the code artefact – partial types and attributes – to express composition.

- The units involved in Hyper/Net composition are fixed. For classes and interfaces, they must all be partials of the same unit type. For methods, they must have the same signature and belong to matching partial types.

Subsection 6.2.3 also points out solutions to achieve adequate hyperslice and hypermodule reuse with Hyper/Net. Solving the first limitative fact requires separating the composition definitions from the code. This can be achieved by implementing composition constructs in an artefact of their own or, at least, in a separate concern of the code artefact. The second limitative fact requires extending Hyper/Net's matching model with more powerful matching

constructs. Here, most MDSoC solutions use regular expressions to identify matching units. These could be adopted for Hyper/Net, but with the adequate care, as Hyper/Net is currently based on type and signature matching. Some of the properties of the current stricter matching model might also be desirable. Recall that Hyper/J lacks the possibility of matching signatures, which can be considered a limitation.

10.3.3 Holistic MDSoC and Hyper/Net

The previous sections focused future work dealing with necessary changes and extensions to Hyper/Net composition support in the code artefact. This section uses a holistic perspective of MDSoC to look at a broader scope of future work.

Still inside the code artefact, each dimension introduces its own perspective of the code body. Hyper/Net does not support virtual decomposition. This way, each dimension will only contain the units that were physically decomposed into it. Virtual decomposition could be interesting in Hyper/Net to provide other useful dimensional perspectives[31]. These perspectives could then be used by developers to work with the units in their physical form, wherever they might exist. Providing such interaction would be an interesting IDE enhancement.

As mentioned in Subsection 4.2.4, the IDE is becoming a central tool in the entire software development process. Thus, adequate support for MDSoC in the IDE is crucial. This motivated the efforts to integrate Hyper/Net with two .NET IDEs (these integrations are presented in Subsections 7.2.1 and 7.2.2). There were two main requirements guiding these integrations:

- Retain the IDE support for one click builds.
- Allow using auto complete and other IDE programming support features with MDSoC decomposed code.

Even though both requirements were supported separately, none of the Hyper/Net IDE integrations supported them simultaneously. The build process implemented by each IDE should be investigated in more detail to search for alternatives that simultaneously realize both requirements of the integration.

Another limitation of the Hyper/Net integration with IDEs is the inexistent error traceability. In the future, these integrations and Hyper/Net itself should be enhanced in order to allow tracing syntax, composition and other errors to the decomposed source code and show them like errors are normally presented in the IDEs. A similar approach is also necessary for debugging.

Typically, MDSoC hyperspaces are defined prior to program compilation. To change the hyperspace of a running program, for instance to add or remove hyperslices, the program must be separately recompiled, stopped and replaced. Adding and removing MDSoC hyperslices from a running program is a valid requirement. Hyper/Net does not support such dynamism, as most other MDSoC implementations do not either. This kind of support would require a different integration approach. Hyper/Net currently works with source code. To be able to change runtime behaviour it would have to work with intermediate code, integrated

[31] With the current Hyper/Net version and the Visual Studio IDE it is possible to navigate the Object dimension in this fashion by using the Visual Studio Class Designer, but no other dimension can be navigated this way.

with the .NET Just-in-time (JIT) compiler. An interesting framework to support this approach is Microsoft Phoenix. It is being developed by Microsoft and consists of a unified framework, offering extension and customization features for the .NET compilers and runtime. A solution with related, yet, less ambitious requirements, that uses Microsoft Phoenix, is discussed for the SetPoint Aspect Oriented Programming implementation [Altman06].

The test artefact is also embraced by the holistic MDSoC perspective. Multi-dimensional unit testing has already been identified, in Section 7.3, as a valuable field and poses interesting challenges for future research. As presented in Section 7.3, testing merged methods provides one such challenge. Further research into the composition of test concerns is required to address this challenge. Eventually, new composition features may be necessary for providing test concerns with the required mechanisms to test standard concern composition, such as merge. More insight into test-driven development and other testing approaches would also be valuable to strengthen the proposed MDSoC testing approach.

Previous sections (7.2, 7.4 and 8.2) already focused situations when a concern needs to use functionality from a different concern. In these cases, it is desirable to avoid introducing a direct dependency between concerns. Future research could explore solutions like defining inter-concern interfaces or using adapters for each different concern that provides the same functionality.

Another interesting field for future research deals with the crossing of boundaries between different hyperspaces. For instance, take one hyperspace defined inside a .NET project, or a set of projects, with its own dimensions and respective concerns. If it is used as a library from a second hyperspace, should it not offer its functionality according to the implemented dimensions and concerns? Eventually, the class library of the first hyperspace should automatically have a set of interfaces generated, one for each concern of its hyperspace. The second hyperspace could then use these interfaces to view the library functionality from the different concern perspectives. The same approach can be thought of in terms of automatic documentation organization.

Partial types are a native language feature of .NET languages that we successfully explored in terms of MDSoC support. It would also be interesting to explore other existing features of software development environments and platforms that can be used for MDSoC. For instance, source control and change management mechanisms could be used together to support a features dimension.

If an MDSoC hyperspace is defined in higher level artefacts, like analysis or design, then, its structure can be automatically generated for the code artefact. This kind of hyperspace traceability between definitions in the different artefacts is essential and should be straightforward to automate given that appropriate analysis and design tools exist.

As IDEs tend to become the tool of choice for all stages in software development, enhancing Hyper/Net IDE integration is an important task for adequate MDSoC support across different artefacts. In fact, as the hyperspace structure is shared between the different artefacts, it should be implemented only once and be shared by the different artefacts of the same hyperspace. This can also be an interesting issue for IDE integration when using design and analysis tools integrated in the IDE.

10.4 Concluding remarks

Software composition addresses relevant problems in Software Engineering. With the advent of subjects, SOP provided a generic modularization mechanism that could be used to overcome limitations with Object Oriented modularization. MDSoC introduced this modularization mechanism into a multi-dimensional structure and extended its context from programming to the entire software development lifecycle. This allowed direct traceability between the different artefacts in software development.

Moreover, both SOP and MDSoC allow the introduction and removal of features without affecting the others. This promotes reuse and also provides support for combining different sets of features through mix-and-match. Mix-and-match can be used to support software product lines.

We developed our own MDSoC implementations so we could use MDSoC in Microsoft .NET. Our .NET partial types approach can be used to implement MDSoC without any supporting software other than a .NET compiler. This comes at the cost of only being able to create very simple MDSoC hyperspaces, but, still, it offers an interesting MDSoC implementation that even supports mix-and-match. To also support the composition of methods, we extended this approach by implementing Hyper/Net. It uses .NET attributes to hold composition information. These attributes are applied directly to the methods, so, Hyper/Net does not separate composition information from the code itself.

Hyper/Net was used to develop some case studies that simultaneously showed the benefits of MDSoC and served to validate our implementations. This validation was consolidated through tests on the functionality of the case studies, using a unit testing approach that we adapted for MDSoC. This validation was not only achieved through case studies but also by analyzing our implementations in the light of the MDSoC model and by comparing them with other MDSoC implementations.

When compared with existing MDSoC implementations, Hyper/Net showed relevant limitations, but these can be addressed and overcome as part of future work. We were able to innovate, introducing the first native MDSoC implementation and by offering support for more than one programming language. Being able to use our MDSoC approaches inside IDEs also simplifies their adoption and leads way for the future support, in the same hyperspace, of the different artefacts that can be manipulated in the IDE.

References

[Altman06] R. Altman, A. Cyment, “Exploring Setpoint: current and future work”. Internal report, available at http://docs.codehaus.org/download/attachments/48359/SetPointWithPhoenix.pdf?version=1, 2006.

[Arsanjani03] A. Arsanjani, B. Hailpern, J. Martin, P. Tarr, "Web Services: Promises and Compromises". Queue vol. 1, pp. 48-58, ACM, April 2003.

[Carver02] L. Carver, "Building Real-World Applications with Aspect-Oriented Modules and Hyper/J". MSc Thesis, University of California, San Diego, 2002.

[Clarke99] S. Clarke, W. Harrison, H. Ossher, P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code". OOPSLA’99, pp. 325-339, ACM Press, 1999.

[CSharp05] "C# Programming Guide". Microsoft, http://msdn2.microsoft.com/en-us/library/67ef8sbd(VS.80).aspx, 2005.

[Dias06] T. Dias, A. Moreira, "Hyper/Net: MDSoC Support for .NET". Workshop of Aspect-Oriented Software Development, JISBD 2006, Sitges, Spain, October 2006.

[Dias07] T. Dias, "Design Patterns & MDSoC: Casos de uso em Software ITS". 3rd Brisa ITS Workshop, São Domingos de Rana, Portugal, February 2007.

[Dijkstra74] E. Dijkstra, "On the role of scientific thought". EWD 447, available at http://www.cs.utexas.edu/users/EWD/index04xx.html, Neuen, The Netherlands, 1974.

[Ecma02] ECMA, "ECMA-335: Common Language Infrastructure (CLI) Partition III: CIL Instruction Set". ECMA (European Association for Standardizing Information and Communication Systems), 2002.

[France03] R. France, G. Georg, I. Ray, "Supporting Multi-Dimensional Separation of Design Concerns". Proceedings of the Third International Workshop on Aspect-Oriented Modeling, 2003.

[González05] C. González, J. Murillo, P. Amaya, "Aspect-oriented analysis: A MDA based approach". In "Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design" edited by Clements et al., March 2005.

[Hailpern01] B. Hailpern, P. Tarr, "Software Engineering for Web Services: A Focus on Separation of Concerns". OOPSLA Workshop on Object-Oriented

Web Services, available as IBM Research Report, RC22184, IBM, 2001.

[Hantelmann06] 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.

[Harrison93] W. Harrison, H. Ossher, "Subject-oriented programming: a critique of pure objects". OOPSLA'93, pp. 411-428, ACM Press, 1993.

[Herrmann00] S. Herrmann, M. Mezini, "On the Need for a Unified MDSOC Model: Experiences from Constructing a Modular Software Engineering Environment". Position paper for the OOPSLA 2000 workshop on Advanced Separation of Concerns, 2000.

[Hill06] P. Hill, S. Holland, R. Laney, "Symmetric Composition of Musical Concerns". AOSD'06 Proceedings, pp. 226-236, ACM Press, 2006.

[Hirschfeld03] R. Hirschfeld, K. Østerbye, M. Wagner, "System Integration Using AOP". In Proceedings of the 3rd Workshop of the German Computer Society (GI) on AOSD, Essen, Germany, 2003.

[Holm03] P. Holm, M. Krüger, B. Spuida, "Dissecting a C# Application: Inside SharpDevelop". Wrox Press, 2003.

[HyperJ03] P. Tarr, H. Ossher, V. Kruskal, M. Kaplan, "Hyper/J website at IBM alphaWorks". http://www.alphaworks.ibm.com/tech/hyperj, IBM, 2003.

[IEEE90] IEEE, "IEEE Standard Glossary of Software Engineering Terminology", Definition for "Software Engineering", pp. 67. IEEE, New York, September 1990.

[Kande00] M. Kandé, A. Strohmeier, "On The Role of Multi-Dimensional Separation of Concerns in Software Architecture". OOPSLA 2000 workshop on Advanced Separation of Concerns, Minneapolis, Minnesota USA, October 2000.

[Kande03] M. Kandé, "A concern-oriented approach to software architecture". PhD Thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2003.

[Lasky03] J. Lasky, J. Sonstein, "Untangling Regulatory Text: Multidimensional Separation of Concerns and Task-Oriented Linking". Presented at ACM HyperText, 2003.

[Liberty01] J. Liberty, "Chapter 18 – Attributes and Reflection". In "Programming C#", 1rst Edition, O'Reilly, 2001.

[Lourenci02] A. Lourenci, "Three-dimensionality emerges out of early aspects". AOSD'02, 2002.

[Lozano06] A. Lozano, M. Wermelinger, B. Nuseibeh, "Degradation archaeology: studying software flaws' evolution". ERCIM Workshop on Software Evolution, 2006.

[Memmert02] J. Memmert, "Designing with Cosmos". Workshop on Aspect Oriented Design, AOSD'02, 2002.

[NRefactory05] NRefactory tutorial video, http://laputa.sharpdevelop.net/NRefactoryTutorialVideo.aspx, 2005.

[NUnit07] NUnit Website, http://www.nunit.org, 2007.

[Ossher96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal, "Specifying subject-oriented composition". Theory and Practice of Object Systems, Vol. 2, Issue 3, pp. 179-202, 1996.

[Ossher99] 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.

[Ossher00] 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.

[Ossher01] H. Ossher, P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software". Comunications of the ACM, Vol. 44, Issue 10, pp. 43-50, October 2001.

[Philippow03] I. Philippow, M. Riebisch, K. Boellert "The Hyper/UML Approach for Feature Based Software Design". Workshop on Modeling With UML, AOSD'03, 2003.

[Rouvellou00] I. Rouvellou, S. Sutton, S. Tai, "Multidimensional Separation of Concerns in Middleware". Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, 22nd ICSE, pp. 106-111, 2000.

[SharpDevelop07] SharpDevelop Website, http://www.sharpdevelop.net/, 2007.

[Stoecker04] M. Stoecker, "Visual Studio 2005 Class Designer". http://msdn2.microsoft.com/en-us/library/aa288743(VS.71).aspx, Visual Studio Technical Articles, Microsoft, May 2004.

[Stuikys02] V. Štuikys, R. Damaševicius, G. Ziberkas, G. Majauskas, "Soft IP Design Framework Using Metaprogramming Techniques". In "Design and Analysis of Distributed Embedded Systems", Kluwer Academic Publishers, pp. 257-266, 2002.

[Sutton02] S. Sutton, P. Tarr, "Aspect-Oriented Design Needs Concern Modeling". Position paper in AOSD'02, 2002.

[Tarr99] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns". 21st ICSE, pp. 107-119, May 1999.

[Tarr01] P Tarr, H. Ossher, "Hyper/J User and Installation Manual". http://www.research.ibm.com/hyperspace, IBM Corporation, 2001.

[Thai03] T. Thai, H. Lam, "Chapter 2 – The Common Language Runtime". In ".NET Framework Essentials", 3rd Edition, O'Reilly, 2003.

[WikiCS] Wikipedia contributors, "C#". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/C_Sharp, September 2007.

[WikiIS] Wikipedia contributors, "IntelliSense". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/IntelliSense, September 2007.

[WikiJa] Wikipedia contributors, "Java (programming language)". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Java_%28programming_language%29, September 2007.

[WikiNL] Wikipedia contributors, ".NET Languages". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Microsoft_.NET_Languages, September 2007.

[WikiNF] Wikipedia contributors, ".NET Framework". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/.NET_Framework, September 2007.

[WikiVB] Wikipedia contributors, "Visual Basic .NET". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Visual_Basic_.NET, September 2007.

Apendix I

Hyper/Net source code

This appendix lists the Hyper/Net source code organized according to the hyperspace presented in Section 8.2. The code is also documented in that same section.

I.1 Hyper/Net Attribute Class Library

I.1.1 Language Features – Bracket Concern

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

  /// <summary>
  /// Defines the bracketing of a method
  /// </summary>
  [AttributeUsage(AttributeTargets.Method, Inherited = true, AllowMultiple
= false)]
  public class MethodBracket : System.Attribute
  {
    public readonly BeforeMethod beforeMethod;
    public readonly AfterMethod afterMethod;
    public readonly string beforeMethodByName;
    public readonly string afterMethodByName;

    /// <summary>
    /// Creates a new method bracket Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    public MethodBracket(BeforeMethod beforeMethod, AfterMethod
afterMethod)
    {
      this.beforeMethod = beforeMethod;
```

```
      this.afterMethod = afterMethod;
    }

    /// <summary>
    /// Creates a new method bracket Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    public MethodBracket(string beforeMethod, string afterMethod)
    {
      this.beforeMethodByName = beforeMethod;
      this.afterMethodByName = afterMethod;
    }
  }
}
```

I.1.2 Language Features – Merge Concern

```
namespace HyperNet
{
  /// <summary>
  /// Distinguishes between different method merge actions
  /// </summary>
  public enum MethodMergeAction { Override, Merge };

  public delegate object MethodMergeResult (params object[]
resultsToMerge);

  /// <summary>
  /// Defines the action used for merging methods
  /// </summary>
  [AttributeUsage(AttributeTargets.Method, Inherited = true, AllowMultiple
= false)]
  public class MethodMerge : System.Attribute
  {
    public readonly MethodMergeAction mergeAction;
    public readonly int priority = -1;
    public readonly MethodMergeResult mergeResult = null;
    public readonly string mergeResultByName = null;

    /// <summary>
    /// Creates a new merge action Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    public MethodMerge(MethodMergeAction mergeAction)
    {
      this.mergeAction = mergeAction;
    }

    /// <summary>
    /// Creates a new merge action Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    /// <param name="priority">Method priority relative to the merged
methods</param>
    public MethodMerge(MethodMergeAction mergeAction, int priority)
    {
      this.mergeAction = mergeAction;
      this.priority = priority;
```

```
    }

    /// <summary>
    /// Creates a new merge action Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    /// <param name="priority">Method priority relative to the merged
methods</param>
    public MethodMerge(MethodMergeAction mergeAction, int priority,
MethodMergeResult mergeResult)
    {
      this.mergeAction = mergeAction;
      this.priority = priority;
      this.mergeResult = mergeResult;
    }
    /// <summary>
    /// Creates a new merge action Attribute
    /// </summary>
    /// <param name="mergeAction">Defines the action used for merging this
method </param>
    /// <param name="priority">Method priority relative to the merged
methods</param>
    public MethodMerge(MethodMergeAction mergeAction, int priority, string
mergeResultByName)
    {
      this.mergeAction = mergeAction;
      this.priority = priority;
      this.mergeResultByName = mergeResultByName;
    }
  }

}
```

I.2 Hyper/Net Console Application

I.2.1 Features – Flow Control

```
namespace HyperNet
{
  partial class MainClass
  {
    public static int Run(string code,
                          SupportedLanguage sourceLang,
                          SupportedLanguage outputLang,
                          string outputFile)
    {
      // Multi-language support
      SupportedLanguage parseLanguage = sourceLang;

      SourceCode source = new SourceCode(code, parseLanguage);

      // Move references to the beginning of the code string
      source.FactorizeImports();

      // Parse the code
      if(!source.Parse())
```

```
      {
        Console.Write("Error while parsing: " + source.ParseErrors);

                string errorFile = outputFile + ".error";
                Console.Write("  Saved Hyper/Net intermediate code to " +
errorFile);
                SaveToFile(errorFile, source.ToString());

                Console.ReadLine();

                return -1;
      }
      else
      {
        //Console.WriteLine("Parse OK.");

        // Merge namespaces
        source.MergeNamespaces();

        // Merge partial types
        source.MergePartialTypes();

        // Merge methods
        source.MergeMethods();

        // Apply bracketing
        source.ApplyBracketing();

        // Get the output in the desired language
        string programOutput = source.GetCode(outputLang);

        // Save to file
        SaveToFile(outputFile, programOutput);

        //Console.WriteLine(programOutput);

                return 0;
      }
    }

    public static int Main(string[] args)
    {
      if(args.Length != 3)
      {
                foreach (string arg in args)
                    Console.WriteLine(arg);

                Console.WriteLine("usage: HyperNet <input project base
directory>\r\n"
                          +"\t<input project file> <output file
name>\r\n\r\n"
                          +@"Example HyperNet.exe
d:\projects\HousePlantEditor\"+"\r\n\t"
                          +@"HousePlant.csproj
d:\projects\HousePlantEditor\Output\HousePlant.cs");
        return -2;
      }

            int ret;

      // TODO: The main try catch approach is only good for a prototype
      try
      {
        // Will contain the merge of the source code
```

```
                string projectBaseDir = args[0] + "\\";
                string projectFile = args[1];
                string targetFile = args[2];

                string merged_source_code =
LoadMergedSourceCode(projectBaseDir, projectFile, targetFile);

                ret = Run(merged_source_code,
                    SupportedLanguage.CSharp,
                    SupportedLanguage.CSharp,
                    targetFile);
      }
      catch(Exception ex)
      {
        Console.WriteLine("Error: " + ex.Message);
        //Console.WriteLine(ex.StackTrace);
        Console.ReadLine();

                ret = -1;
      }

            return ret;
    }
    }
}
```

I.2.2 Features – Input

```
using ICSharpCode.SharpDevelop.Project;

namespace HyperNet
{
  partial class MainClass
  {
        private static string LoadMergedSourceCode(string projectBaseDir,
string projectFile, string targetFile)
        {
            // Will contain the merge of the source code
            string merged_source_code = "";

            // Get all source files from the project file
            FileStream projFile = File.Open(projectBaseDir + projectFile,
FileMode.Open);

            /* Not used: XmlDocument approach for Project file editing:
            XmlDocument doc = new XmlDocument();
            doc.Load(projFile);

            XmlNodeList compiles = doc.GetElementsByTagName("Compile");

            foreach (XmlNode compile in compiles)
            {
                // Read the source file content
                FileStream sourceFile =
                     File.Open(projectBaseDir +
compile.Attributes["Include"].InnerText, FileMode.Open);
                StreamReader sr = new StreamReader(sourceFile);
                merged_source_code += sr.ReadToEnd();
                sr.Close();
                sourceFile.Close();
            }
```

```
            NameXmlElements(doc, compiles, "EmbeddedResource_Temp");
            XmlNodeList embeddeds =
doc.GetElementsByTagName("EmbeddedResource");
            NameXmlElements(doc, embeddeds, "Compile");
            XmlNodeList compiles_temp =
doc.GetElementsByTagName("EmbeddedResource_Temp");
            NameXmlElements(doc, compiles_temp, "EmbeddedResource");

            projFile.Seek(0, SeekOrigin.Begin);
            doc.Save(projFile);
            //doc.Save(File.Create(projectBaseDir + projectFile +
".hn.csproj"));
             */

            XmlReader projReader = XmlReader.Create(projFile);
            while (projReader.Read())
            {
                if (projReader.NodeType == XmlNodeType.Element
                   && projReader.Name.Equals("EmbeddedResource"))
                {
                    FileStream sourceFile =
                        File.Open(projectBaseDir +
projReader.GetAttribute("Include"), FileMode.Open);
                    StreamReader sr = new StreamReader(sourceFile);
                    merged_source_code += sr.ReadToEnd();
                    sr.Close();
                    sourceFile.Close();
                }
            }
            projReader.Close();

            projFile.Close();

            // Save a copy of the merged source code (for debug purposes)
            SaveToFile(targetFile + ".original", merged_source_code);

            return merged_source_code;
        }

        /*
        private static void NameXmlElements(XmlDocument doc, XmlNodeList
nodes, string newName)
        {
            for (int i = nodes.Count - 1; i >= 0; i--)
            {
                XmlNode node = nodes[i];

                // Replace the element tag
                XmlElement compToEmbedded = doc.CreateElement(newName,
node.NamespaceURI);
                foreach (XmlAttribute att in node.Attributes)

compToEmbedded.Attributes.Append((XmlAttribute)att.Clone());
                foreach (XmlNode child in node.ChildNodes)
                {
                    XmlElement childElem = doc.CreateElement(child.Name,
child.NamespaceURI);
                    childElem.InnerText = child.InnerText;
                    compToEmbedded.PrependChild(childElem);
                }
                node.ParentNode.ReplaceChild(compToEmbedded, node);
            }
        }
```

```
        */
  }
}
```

I.2.3 Features – Kernel

```
namespace HyperNet
{
  public partial class SourceCode
  {
    string sourceCode;
    public readonly SupportedLanguage Language;

    /* Overriden in Feature.Output
    public string Code
    {
      get { return sourceCode; }
    }
    */

    public SourceCode(string sourceCode, SupportedLanguage language)
    {
      this.sourceCode = sourceCode;
      this.Language = language;
    }

        public override string ToString()
        {
            return this.sourceCode;
        }
  }
}
```

I.2.4 Features – Output

```
namespace HyperNet
{
  partial class MainClass
  {
    private static void SaveToFile(string fName, string code)
    {
      if(fName != null)
      {
        FileStream outFile = File.Open(fName, FileMode.Create);
        StreamWriter outSw = new StreamWriter(outFile);
        outSw.Write(code);
        outSw.Close();
        outFile.Close();
      }
    }
  }
}

namespace HyperNet
{
  public partial class SourceCode
  {
    public string Code
```

```
    {
      get
      {
        if(this.parser != null)
          return GetCode(this.Language);
        else
          return sourceCode;
      }
    }
  }
}
```

I.2.5 Features – Parse Preparations

```
namespace HyperNet
{
  public partial class SourceCode
  {
    public void FactorizeImports()
    {
      Hashtable referenceHT = new Hashtable();

      string searchStr = "using ";
      string delimiter = ";";

      if(this.Language == SupportedLanguage.VBNet)
      {
        searchStr = "Imports ";
        delimiter = "\n";
      }

      int refIndex, lastFound = 0;
            while ((refIndex = this.sourceCode.IndexOf(searchStr,
lastFound)) != -1)
            {
                int endRefIndex = this.sourceCode.IndexOf(delimiter,
refIndex + searchStr.Length);
                if (endRefIndex - refIndex + 1 > 0)
                {
                    string reference = this.sourceCode.Substring(refIndex,
endRefIndex - refIndex + 1);
                    //reference.Replace(" ", "");

                    this.sourceCode = this.sourceCode.Substring(0,
Math.Max(refIndex - 1, 0))
                        + this.sourceCode.Substring(endRefIndex + 1);

                    if (!referenceHT.Contains(reference))
                        referenceHT.Add(reference, reference);
                }
                else
                    lastFound = refIndex + 1;
            }

      string allDistinctReferences = "";

      foreach(string reference in referenceHT.Keys)
      {
        allDistinctReferences += reference;
      }
```

```
      this.sourceCode = allDistinctReferences + this.sourceCode;
    }
  }
}
```

I.2.6 Features – Parsing

```
using ICSharpCode.NRefactory.Parser;
using ICSharpCode.NRefactory.Parser.AST;
using ICSharpCode.NRefactory.PrettyPrinter;

namespace HyperNet
{
  public partial class SourceCode
  {
    IParser parser = null;

    public string ParseErrors
    {
      get
      {
        if(parser != null)
          return parser.Errors.ErrorOutput;
        else
          return null;
      }
    }

    public bool Parse()
    {
      StringReader sr = new StringReader(this.Code);
      parser = ParserFactory.CreateParser(this.Language, sr);
      parser.Parse();

      if(parser.Errors.count > 0)
        return false;
      else
        return true;
    }

        public string GetCode(SupportedLanguage outputLanguage)
        {
            if (parser == null)
            {
                throw new Exception("Parser not initialized, no code can be
generated.");
            }
            else
            {
                IOutputASTVisitor outputVis;
                if (outputLanguage == SupportedLanguage.CSharp)
                    outputVis = new CSharpOutputVisitor();
                else
                    outputVis = new VBNetOutputVisitor();

                this.parser.CompilationUnit.AcceptVisitor(outputVis, null);
                return outputVis.Text;
            }
        }
    }
}
```

I.2.7 Features – Namespace Composition

```
using ICSharpCode.NRefactory.Parser;
using ICSharpCode.NRefactory.Parser.AST;

namespace HyperNet
{
  public partial class SourceCode
  {
    /// <summary>
    /// Merge all repeated namespaces ocurring in the parse tree
    /// </summary>
    public void MergeNamespaces()
    {
      Hashtable nsHT = new Hashtable();
      List<NamespaceDeclaration> remNSList = new
List<NamespaceDeclaration>();

      foreach(INode inode in this.parser.CompilationUnit.Children)
      {
        if(inode is NamespaceDeclaration)
        {
          NamespaceDeclaration ns = (NamespaceDeclaration)inode;
          if(!nsHT.Contains(ns.Name))
          {
            nsHT.Add(ns.Name, ns);
          }
          else
          {
            NamespaceDeclaration mainNS =
(NamespaceDeclaration)nsHT[ns.Name];
            foreach(INode node in ns.Children)
            {
              mainNS.AddChild(node);
            }

            remNSList.Add(ns);
          }
        }
      }

      foreach(NamespaceDeclaration ns in remNSList)
      {
        this.parser.CompilationUnit.Children.Remove(ns);
      }
    }
  }
}
```

I.2.8 Features – Partial Type Composition

```
using ICSharpCode.NRefactory.Parser;
using ICSharpCode.NRefactory.Parser.AST;

namespace HyperNet
{
  public partial class SourceCode
  {
        private List<AttributeSection>
MergeAttributeSections(List<AttributeSection> ass1, List<AttributeSection>
ass2)
```

```
        {
            List<AttributeSection> ret = new List<AttributeSection>();

            foreach (AttributeSection as1 in ass1)
            {
                AttributeSection asNew = new
AttributeSection(as1.AttributeTarget, new
List<ICSharpCode.NRefactory.Parser.AST.Attribute>());

                foreach (ICSharpCode.NRefactory.Parser.AST.Attribute at1 in
as1.Attributes)
                {
                    bool alreadyPresent = false;
                    foreach (AttributeSection as2 in ass2)
                    {
                        foreach
(ICSharpCode.NRefactory.Parser.AST.Attribute at2 in as2.Attributes)
                        {
                            if (at2.Name == at1.Name)
                                alreadyPresent = true;
                        }
                    }
                    if(!alreadyPresent)
                        asNew.Attributes.Add(at1);
                }

                if (asNew.Attributes.Count > 0)
                    ret.Add(asNew);
            }

            foreach (AttributeSection as2 in ass2)
            {
                ret.Add(as2);
            }

            return ret;
        }

    public void MergePartialTypes()
    {
      foreach(INode inode in parser.CompilationUnit.Children)
      {
        if(inode is NamespaceDeclaration)
        {
                    Hashtable ptHT = new Hashtable();
                    List<TypeDeclaration> remPTList = new
List<TypeDeclaration>();

                    foreach (INode nsNode in inode.Children)
          {
            if(nsNode is TypeDeclaration)
            {
              TypeDeclaration td = (TypeDeclaration)nsNode;

              if(td.Modifier.CompareTo(Modifier.Partial) >= 0)
              {
                if(!ptHT.Contains(td.Name))
                {
                  ptHT.Add(td.Name, td);
                  td.Modifier -= Modifier.Partial;
                }
                else
                {
                  TypeDeclaration mainTD = (TypeDeclaration)ptHT[td.Name];
```

```
                    foreach(INode node in td.Children)
                    {
                      mainTD.AddChild(node);
                    }

                    foreach(TypeReference tr in td.BaseTypes)
                    {
                      bool referenceExists = false;
                      foreach(TypeReference existingTr in mainTD.BaseTypes)
                      {
                        if(existingTr.SystemType.Equals(tr.SystemType))
                        {
                          referenceExists = true;
                          break;
                        }
                      }
                      if(!referenceExists)
                        mainTD.BaseTypes.Add(tr);
                    }

                                        // Copy attributes
                                        mainTD.Attributes =
this.MergeAttributeSections(td.Attributes, mainTD.Attributes);

                    td.Modifier -= Modifier.Partial;
                    mainTD.Modifier |= td.Modifier;

                    remPTList.Add(td);
                  }
                }
              }
            }

            foreach(TypeDeclaration td in remPTList)
            {
              inode.Children.Remove(td);
            }
          }
        }
      }
    }
}
```

I.2.9 Language Features – Bracket Concern

```
using ICSharpCode.NRefactory.Parser;
using ICSharpCode.NRefactory.Parser.AST;

namespace HyperNet
{
  public partial class SourceCode
  {
    public void InsertInMethodBody(MethodDeclaration method, int pos, INode
node)
    {
      if(pos >= method.Body.Children.Count)
        method.Body.AddChild(node);
      else
      {
        List<INode> list = new List<INode>();
```

```
        int numElemsBody = method.Body.Children.Count;

        for(int i = numElemsBody - 1; i >= pos; i--)
        {
          list.Add(method.Body.Children[i]);
          method.Body.Children.RemoveAt(i);
        }

        method.Body.AddChild(node);

        for(int i = list.Count - 1; i >= 0; i--)
        {
          method.Body.AddChild(list[i]);
          list.RemoveAt(i);
        }
      }
    }


    /// <summary>
    /// Applies bracketing
    /// </summary>
    public void ApplyBracketing()
    {
      foreach(INode inode in parser.CompilationUnit.Children)
      {
        if(inode is NamespaceDeclaration)
        {
          foreach(INode nsNode in inode.Children)
          {
            if(nsNode is TypeDeclaration)
            {
              TypeDeclaration td = (TypeDeclaration)nsNode;

              foreach(INode classNode in td.Children)
              {
                if(classNode is MethodDeclaration)
                {
                  MethodDeclaration method = (MethodDeclaration)classNode;

                  // Get the method bracketing attributes (if any)
                  MethodBracket bracketAtt =
                    (MethodBracket)GetAttribute(method,
typeof(MethodBracket));

                  // Remove the attribute
                  this.RemoveAttribute(method, typeof(MethodBracket));

                  if(bracketAtt != null)
                  {
                    string baseMethodInfoID = "_method_BasicInfo";
                    if(bracketAtt.beforeMethodByName != null
                       || bracketAtt.afterMethodByName != null)
                    {
                      FieldReferenceExpression getCurrentMethod =
                        new FieldReferenceExpression(
                                                    new
FieldReferenceExpression(

new FieldReferenceExpression(

new IdentifierExpression(

"System"), "Reflection"), "MethodBase"), "GetCurrentMethod");
```

```
                        InvocationExpression getCurrentMethod_Invocation =
                          new InvocationExpression(getCurrentMethod, null);
                        VariableDeclaration varDeclaration =
                          new VariableDeclaration(baseMethodInfoID,

getCurrentMethod_Invocation,
                                                  new
TypeReference("System.Reflection.MethodBase"));

                        InsertInMethodBody(method, 0, new
LocalVariableDeclaration(varDeclaration));
                      }

                      if(bracketAtt.beforeMethodByName != null)
                      {
                        FieldReferenceExpression methodName =
                          new FieldReferenceExpression(new
ThisReferenceExpression(),

bracketAtt.beforeMethodByName);
                        InvocationExpression ie = new
InvocationExpression(methodName, null);

                        ie.Arguments.Add(new
IdentifierExpression(baseMethodInfoID));
                        foreach (ParameterDeclarationExpression param in
method.Parameters)
                        {
                          Expression expr = new
IdentifierExpression(param.ParameterName);
                          ie.Arguments.Add(expr);
                        }

                        InsertInMethodBody(method, 1, new
StatementExpression(ie));
                      }

                      if(bracketAtt.afterMethodByName != null)
                      {
                        int num_return_stmt = 0;
                        foreach(INode metNode in method.Body.Children)
                        {
                          if(metNode is ReturnStatement)
                          {
                            num_return_stmt++;
                            ReturnStatement returnStmt =
(ReturnStatement)metNode;

                            Expression retVal = returnStmt.Expression;
                            FieldReferenceExpression methodName =
                              new FieldReferenceExpression(new
ThisReferenceExpression(),

bracketAtt.afterMethodByName);
                            InvocationExpression ie = new
InvocationExpression(methodName, null);

                            ie.Arguments.Add(new
IdentifierExpression(baseMethodInfoID));
                            ie.Arguments.Add(retVal);
                            foreach (ParameterDeclarationExpression param in
method.Parameters)
                            {
```

```
                            Expression expr = new
IdentifierExpression(param.ParameterName);
                            ie.Arguments.Add(expr);
                          }

                          returnStmt.Expression = new
CastExpression(method.TypeReference, ie, CastType.Cast);
                        }
                      }

                      if(num_return_stmt == 0)
                      {
                        FieldReferenceExpression methodName =
                          new FieldReferenceExpression(new
ThisReferenceExpression(),

bracketAtt.afterMethodByName);
                        InvocationExpression ie = new
InvocationExpression(methodName, null);

                        ie.Arguments.Add(new
IdentifierExpression(baseMethodInfoID));
                        ie.Arguments.Add(new PrimitiveExpression(null,
"null"));
                        foreach (ParameterDeclarationExpression param in
method.Parameters)
                        {
                          Expression expr = new
IdentifierExpression(param.ParameterName);
                          ie.Arguments.Add(expr);
                        }

                        method.Body.AddChild(new StatementExpression(ie));
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

I.2.10 Language Features – Merge Concern

```
using ICSharpCode.NRefactory.Parser;
using ICSharpCode.NRefactory.Parser.AST;

namespace HyperNet
{
  /*
  public class Invocation : IComparable
  {
    public readonly int Priority;
    public readonly VariableDeclaration Variable;

    public Invocation(int priority, VariableDeclaration variable)
    {
```

```
      this.Priority = priority;
      this.Variable = variable;
    }

    public int CompareTo(object inv2)
    {
      if(inv2 is Invocation)
        return this.Priority.CompareTo(((Invocation)inv2).Priority);
      else
        return -1;
    }
  }
   */

  public partial class SourceCode
  {
        private string GetAssemblyQualifiedName(string typeName, Type
assemblyType)
        {
            return typeName + "," + assemblyType.Assembly.FullName;
        }

    private InvocationExpression CreateMethodCall(MethodDeclaration method)
    {
      FieldReferenceExpression methodName = new
FieldReferenceExpression(new ThisReferenceExpression(),

method.Name);
      InvocationExpression ie = new InvocationExpression(methodName, null);
      foreach (ParameterDeclarationExpression param in method.Parameters)
      {
        Expression expr = new IdentifierExpression(param.ParameterName);
        if (param.ParamModifier == ParamModifier.Ref)
        {
          expr = new DirectionExpression(FieldDirection.Ref, expr);
        }
        ie.Arguments.Add(expr);
      }
      return ie;
    }

    private bool EqualMethodSignatures(MethodDeclaration method1,
MethodDeclaration method2)
    {

if(!method1.TypeReference.SystemType.Equals(method2.TypeReference.SystemTyp
e))
        return false;

      if(method1.Parameters.Count != method2.Parameters.Count)
        return false;

      for(int i = 0; i < method1.Parameters.Count; i++)
      {

if(!method1.Parameters[i].TypeReference.SystemType.Equals(method2.Parameter
s[i].TypeReference.SystemType))
          return false;
      }

      return true;
    }

    /// <summary>
```

```
    /// Returns the first attribute found with the specified type
    /// </summary>
    /// <param name="method">Method declaration to be searched</param>
    /// <returns></returns>
    private object GetAttribute(MethodDeclaration method, Type
requiredAttributeType)
    {
      foreach(AttributeSection aSec in method.Attributes)
      {
        foreach(ICSharpCode.NRefactory.Parser.AST.Attribute att in
aSec.Attributes)
        {
                    // Hyper/Net attributes belong to a different Assembly.
Need to search for these types there.
                    string attAssemblyQualifiedName =
GetAssemblyQualifiedName(att.Name, typeof(MethodMerge));

                    Type attType = Type.GetType(attAssemblyQualifiedName);
                    if (attType != null &&
attType.Equals(requiredAttributeType))
          {
            ArrayList list = new ArrayList();
            foreach(Expression arg in att.PositionalArguments)
            {
              if(arg is FieldReferenceExpression)
              {
                FieldReferenceExpression fre =
(FieldReferenceExpression)arg;

                string argType;
                string argValue;
                if(fre.TargetObject is IdentifierExpression)
                {
                  argType =
((IdentifierExpression)fre.TargetObject).Identifier;
                  argValue = fre.FieldName;
                }
                else if(fre.TargetObject is FieldReferenceExpression)
                {
                  Expression targetExp = fre.TargetObject;
                  argType = "";
                  argValue = fre.FieldName;
                  while(targetExp is FieldReferenceExpression)
                  {
                    FieldReferenceExpression internalFre =
(FieldReferenceExpression)targetExp;

                    argType = internalFre.FieldName +
                      (argType.Equals(String.Empty) ? "" : "." + argType);

                    targetExp = internalFre.TargetObject;
                  }

                  if(targetExp is IdentifierExpression)
                    argType = ((IdentifierExpression)targetExp).Identifier
+ "." + argType;
                  else
                    throw new Exception("Internal error while getting
attribute hierarchy.");
                }
                else
                {
                  throw new Exception("Error in attribute "+attType+"
definition with "+fre+".");
```

```
                }

                                // Hyper/Net attributes belong to a
different Assembly. Need to search for these types there.
                                string argAssemblyQualifiedName =
GetAssemblyQualifiedName(argType, typeof(MethodMergeAction));
                                Type argRealType =
Type.GetType(argAssemblyQualifiedName);
                if(argRealType == null)
                  throw new Exception("Cannot find the Type '"+argType+"'
of a parameter for "+requiredAttributeType);

                if(argRealType.IsEnum)
                {
                  list.Add(Enum.Parse(argRealType, argValue));
                }
                else
                {
                  Console.WriteLine(argValue);
                  Console.WriteLine(argRealType);

                  string[] parameters = { argValue };
                  Type[] parameterTypes = { argValue.GetType() };

                  MethodInfo parseMethod = argRealType.GetMethod("Parse",
parameterTypes);
                  list.Add(parseMethod.Invoke(null, parameters));
                }
              }
              else if(arg is PrimitiveExpression)
              {
                PrimitiveExpression prim = (PrimitiveExpression)arg;

                list.Add(prim.Value);
              }
              else if(arg is UnaryOperatorExpression
                      && ((UnaryOperatorExpression)arg).Expression is
PrimitiveExpression)
              {
                UnaryOperatorExpression opExp =
(UnaryOperatorExpression)arg;
                PrimitiveExpression prim =
(PrimitiveExpression)opExp.Expression;

                // TODO: Change the method of processing operator types
                if(opExp.Op == UnaryOperatorType.Minus
                   && prim.Value.GetType().Equals(typeof(int)))
                  list.Add(- (int)prim.Value);
                else if(opExp.Op == UnaryOperatorType.Not
                        && prim.Value.GetType().Equals(typeof(bool)))
                  list.Add(! (bool)prim.Value);
                else
                  list.Add(prim.Value);
              }
              // this is a literal, the attribute should provide a
constructor with a string instead
              else if(arg is IdentifierExpression)
              {
                IdentifierExpression ie  = (IdentifierExpression)arg;
                list.Add(ie.Identifier);
              }
              else
              {
```

```
                Console.WriteLine("Unexpected attributte argument: " + arg
+ ".");
              }
            }

            return Activator.CreateInstance(requiredAttributeType,
list.ToArray());
          }
        }
      }

      return null;
    }

    /// <summary>
    /// Remove all attributes found with the specified type
    /// </summary>
    /// <param name="method">Method declaration to be searched</param>
    private void RemoveAttribute(MethodDeclaration method, Type
requiredAttributeType)
    {
      List<AttributeSection> remASecList =
        new List<AttributeSection>();
      foreach(AttributeSection aSec in method.Attributes)
      {
        List<ICSharpCode.NRefactory.Parser.AST.Attribute> remAttList =
          new List<ICSharpCode.NRefactory.Parser.AST.Attribute>();
        foreach(ICSharpCode.NRefactory.Parser.AST.Attribute att in
aSec.Attributes)
        {
                    // Hyper/Net attributes belong to a different Assembly.
Need to search for these types there.
                    string assemblyQualifiedName = att.Name + "," +
typeof(MethodMerge).Assembly.FullName;
                    Type attType = Type.GetType(assemblyQualifiedName);
          if(attType != null && attType.Equals(requiredAttributeType))
          {
            remAttList.Add(att);
          }
        }

        foreach(ICSharpCode.NRefactory.Parser.AST.Attribute att in
remAttList)
          aSec.Attributes.Remove(att);

        remAttList.Clear();

        if(aSec.Attributes.Count == 0)
          remASecList.Add(aSec);
      }

      foreach(AttributeSection aSec in remASecList)
        method.Attributes.Remove(aSec);
    }

    /// <summary>
    /// Merges methods
    /// </summary>
    public void MergeMethods()
    {
      foreach(INode inode in parser.CompilationUnit.Children)
      {
        if(inode is NamespaceDeclaration)
        {
```

```
          foreach(INode nsNode in inode.Children)
          {
            if(nsNode is TypeDeclaration)
            {
              Hashtable metHT = new Hashtable();
              List<MethodDeclaration> remMetList = new
List<MethodDeclaration>();
              List<MethodDeclaration> addMetList = new
List<MethodDeclaration>();

              TypeDeclaration td = (TypeDeclaration)nsNode;

              foreach(INode classNode in td.Children)
              {
                if(classNode is MethodDeclaration)
                {
                  MethodDeclaration method = (MethodDeclaration)classNode;

                  bool hasOverride = false;
                  bool hasMerge = false;
                  string originalMethodName = method.Name;


                  // Step 1: Search for methods with the same name, at the
same level
                  //         and determine the action for Step 2
                  int conflictNum = 0;
                  foreach(INode neighbourClassNode in td.Children)
                  {
                    if(neighbourClassNode is MethodDeclaration
                       && (neighbourClassNode == classNode
                           ||
(((MethodDeclaration)neighbourClassNode).Name.Equals(originalMethodName)
                               &&
EqualMethodSignatures((MethodDeclaration)neighbourClassNode, method))))
                    {
                      MethodDeclaration neighbourMethod =
                        ((MethodDeclaration)neighbourClassNode);
                      conflictNum++;

                      // Get the method merge attributes (if any)
                      MethodMerge mergeAtt =
                        (MethodMerge)GetAttribute(neighbourMethod,
typeof(MethodMerge));

                      if(mergeAtt != null)
                      {
                        if(mergeAtt.mergeAction == MethodMergeAction.Merge)
                          hasMerge = true;
                        else if(mergeAtt.mergeAction ==
MethodMergeAction.Override)
                        {
                          if(hasOverride)
                            throw new Exception(
                                                "More than one Override
merge action defined for method "
                                                + method.Name);
                          hasOverride = true;
                        }
                      }
                    }
                  }

                  if(hasOverride && hasMerge)
```

```
                    throw new Exception("Method " + method.Name + " has
both overrides and merges defined.");
                  else if(conflictNum == 1 && hasMerge)
                  {
                    RemoveAttribute(method, typeof(MethodMerge));
                  }
                  else if(hasOverride)
                  {
                    // Step 2: Remove all methods but the Override one
                    foreach(INode neighbourClassNode in td.Children)
                    {
                      if(neighbourClassNode is MethodDeclaration
                         && (neighbourClassNode == classNode
                             ||
(((MethodDeclaration)neighbourClassNode).Name.Equals(originalMethodName)
                                 &&
EqualMethodSignatures((MethodDeclaration)neighbourClassNode, method))))
                      {
                        MethodDeclaration neighbourMethod =
                          ((MethodDeclaration)neighbourClassNode);

                        // Get the method merge attributes (if any)
                        MethodMerge mergeAtt =
                          (MethodMerge)GetAttribute(neighbourMethod,
typeof(MethodMerge));

                        if(mergeAtt == null || mergeAtt.mergeAction !=
MethodMergeAction.Override)
                        {
                          if(!remMetList.Contains(neighbourMethod))
                            remMetList.Add(neighbourMethod);
                        }
                        else
                        {
                          //RemoveAttribute(neighbourMethod,
typeof(MethodMerge));
                        }
                      }
                    }
                  }
                  else if(conflictNum > 1 && hasMerge)
                  {
                    conflictNum = 0;

                    // A new method must be created, merging each of the
existing methods
                    MethodDeclaration md = new
MethodDeclaration(originalMethodName,

method.Modifier,

method.TypeReference,

method.Parameters,
                                                                    null);
                    md.Body = new BlockStatement();

                    string resultMethodName = null;

                    System.Collections.Generic.SortedList<int,
VariableDeclaration> varsList
                      = new System.Collections.Generic.SortedList<int,
VariableDeclaration>();
```

```
                    System.Collections.Generic.SortedList<int, INode>
invokeList
                      = new System.Collections.Generic.SortedList<int,
INode>();

                    // Step 2: Rename each method and make them be invoked
by a new method
                    foreach(INode neighbourClassNode in td.Children)
                    {
                      if(neighbourClassNode is MethodDeclaration
                         && (neighbourClassNode == classNode
                             ||
(((MethodDeclaration)neighbourClassNode).Name.Equals(originalMethodName)
                                 &&
EqualMethodSignatures((MethodDeclaration)neighbourClassNode, method))))
                       {
                        MethodDeclaration neighbourMethod =
                          ((MethodDeclaration)neighbourClassNode);
                        conflictNum++;

                        // Rename merged method
                        neighbourMethod.Name = neighbourMethod.Name + "_" +
conflictNum;

                        // Change modifier
                        neighbourMethod.Modifier = Modifier.Private;

                        // Invoke each partial method
                        InvocationExpression call =
CreateMethodCall(neighbourMethod);

                        // Get the method merge attributes (if any)
                        MethodMerge mergeAtt =
                          (MethodMerge)GetAttribute(neighbourMethod,
typeof(MethodMerge));
                        int priority = -2;
                        if(mergeAtt != null)
                        {
                          priority = mergeAtt.priority;
                          if(mergeAtt.mergeResultByName != null)
                            resultMethodName = mergeAtt.mergeResultByName;
                        }

                        // Compose the invocation with return value keeping
                        INode invokeDecl;
                        if(!method.TypeReference.ToString().Equals("void"))
                        {
                          // Declare variable to hold each partial result
                          VariableDeclaration varDeclaration =
                            new VariableDeclaration("res_"+conflictNum,
                                                    call,
                                                    method.TypeReference);
                          // Add the variable to the list for returning
calculation
                          try
                          {
                            varsList.Add(-priority, varDeclaration);
                          }
                          catch(Exception)
                          {
                            throw new Exception("MethodMerge: Adding
repeated priority entries.");
                          }
```

```
                        invokeDecl = new
LocalVariableDeclaration(varDeclaration);
                    }
                    else
                    {
                      invokeDecl = new StatementExpression(call);
                    }

                    // Add the invocation to the list for final order
addition to the body
                    invokeList.Add(-priority, invokeDecl);

                    // Remove HyperNet's own attributes
                    RemoveAttribute(neighbourMethod,
typeof(MethodMerge));

                    // Copy existing attributes to the new method
                    foreach(AttributeSection asec in
neighbourMethod.Attributes)
                    {

foreach(ICSharpCode.NRefactory.Parser.AST.Attribute att
                                in asec.Attributes)
                      {
                        bool containsAtt = false;
                        foreach(AttributeSection mdASec in
md.Attributes)
                        {

foreach(ICSharpCode.NRefactory.Parser.AST.Attribute mdAtt
                                    in asec.Attributes)
                          {
                            // TODO: Obviously this check must ensure
mutch more,
                            //       the attribute itself should be
comparable
                            if(mdAtt.Children.Count ==
att.Children.Count
                               && mdAtt.Name.Equals(att.Name)
                              )
                            {
                              containsAtt = true;
                              break;
                            }
                          }
                        }
                        if(!containsAtt)
                        {
                          if(md.Attributes.Count < 1)
                            md.Attributes.Add(new AttributeSection("",
                                                                   new
List<ICSharpCode.NRefactory.Parser.AST.Attribute>()));
                          md.Attributes[0].Attributes.Add(att);
                        }
                      }
                    }

                    // Remove all attributes from the merged method
                    neighbourMethod.Attributes.Clear();
                  }
                }

                // Add method invocation in order
                foreach(INode var in invokeList.Values)
```

```
                    {
                      md.Body.AddChild(var);
                    }

                    if(resultMethodName == null)
                    {
                      //ReturnStatement rs = new ReturnStatement(I
                    }
                    else
                    {
                      // Add the return calculation method
                      FieldReferenceExpression methodName =
                        new FieldReferenceExpression(new
ThisReferenceExpression(),
                                                           resultMethodName);
                      InvocationExpression ie =
                        new InvocationExpression(methodName, null);
                      foreach(VariableDeclaration var in varsList.Values)
                      {
                        Expression expr = new
IdentifierExpression(var.Name);
                        ie.Arguments.Add(expr);
                      }

                      CastExpression ce = new
CastExpression(method.TypeReference, ie, CastType.Cast);

                      ReturnStatement rs = new ReturnStatement(ce);
                      md.Body.AddChild(rs);
                    }

                    // Add the new method for latter adding to the class
                    addMetList.Add(md);
                  }
                  else if(conflictNum > 1)
                  {
                    throw new Exception("Method "+method.Name+" requires at
least one MergeMethod definition (Merge or Override).");
                  }
                }
              }

              foreach(MethodDeclaration md in remMetList)
              {
                nsNode.Children.Remove(md);
              }

              foreach(MethodDeclaration md in addMetList)
              {
                nsNode.Children.Add(md);
              }

              // HyperNet attributes clean-up:
                            // TODO: This shouldn't be necessary here...
              foreach(INode neighbourClassNode in td.Children)
              {
                if(neighbourClassNode is MethodDeclaration)
                {
                  MethodDeclaration method =
                    ((MethodDeclaration)neighbourClassNode);

                  RemoveAttribute(method, typeof(MethodMerge));
                }
              }
```

```
                    }
                }
            }
        }
    }
}
```