JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 4, Mai-June 2006

# ADDING ASPECT-ORIENTED PROGRAMMING FEATURES TO C#.NET BY USING MULTIDIMENSIONAL SEPARATION OF CONCERNS (MDSOC) APPROACH

**Angela Hantelmann** and **Cui Zhang**, Department of Computer Science, California State University, Sacramento, CA 95819-6021

### Abstract

Aspect Oriented Programming (AOP) methodology is based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to automatically weave them together into a coherent program. The term Aspect Oriented Programming includes Multidimensional Separation of Concerns, Subject Oriented Programming, Adaptive Programming and Composition Filters. Multidimensional Separation of Concerns (MDSOC) addresses the issues of software evolution and permits the encapsulation of various kinds of concerns simultaneously (even when concerns were not anticipated originally) and the integration of separate concerns. HyperC# tool supports Multidimensional Separation of Concerns for C#. Adopting some of the techniques from Hyper/J (which is MDSOC for Java), HyperC# tool demonstrates how the AOP extension to C# can enhance software evolution, modularity and reusability.

## 1 INTRODUCTION

In Object-Oriented (OO) programming, a software system is built from a collection of classes. Each of these classes has a well defined task. In an OO application, those classes collaborate to achieve the application's overall goal. But, there are parts of a system that cannot be viewed as being the responsibility of only one class; they cross cut the entire system and affect many classes. Examples might be locking in a distributed application, exception handling, or logging method calls. Of course, the code that handles these parts can be added to each class, but that would defy the principle that each class has well-defined responsibilities. This is where Aspect-Oriented Programming (AOP) comes into play. AOP defines an *aspect*, which is used to encapsulate crosscutting aspects of a software system in a separate module. In this way, the application classes keep their well-defined responsibilities. Additionally, each aspect captures a crosscutting behavior.

Cite this column as follows: Angela Hantelmann, Cui Zhang: “Adding Aspect-Oriented Programming Features to C#.NET by using Multidimensional Seperation of Concerns (MDSOC) Approach”, in *Journal of Object Technology*, vol. 5 no. 4 Mai-June 06, pp. 59-83 http://www.jot.fm/issues/issue_2006_05/article1

AOP has been implemented with several approaches. Those approaches include: MDSOC [6], A Layered Approach [5], Reflexion and Meta-Object Protocols [9], Adaptive Programming [4], and Composition Filters [1]. In this paper the MDSOC approach is chosen. Multidimensional Separation of Concerns (MDSOC) addresses the issues of software evolution and permits identification, effective encapsulation of arbitrary kinds of concerns simultaneously (even when concerns were not anticipated originally), and the integration of separate concerns.

There are not many AOP systems available implemented using MDSOC. Those are Hyper/J [8] and Hyper-VB [3]. Hyper/J is a tool that supports "multi-dimensional" separation and integration of concerns in standard Java software. Hyper-VB is a tool that supports "multi-dimensional" separation of concern and integration of concerns in Visual Basic .NET.

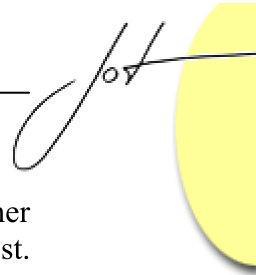
C# is a new Microsoft OOP language which is part of the .NET platform. This paper presents the HyperC# tool, implemented using C#. With HyperC#, a graphical user interface system is created that provides AOP features to C# developers or programmers by using MDSOC approach. Using this tool, C #developers can create new concern classes and new hypermodules. Using integration relationships, the system integrates concern classes together by using the pre-processor developed to create a new C# class that can be further used in other integration files, or can be compiled in an executable file using the C# compiler. HyperC# System demonstrates how the AOP extension to C# can enhance software evolution, modularity and reusability.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Multidimensional Separation of Concerns

The primary goals of software engineering are to improve software quality, to reduce costs of software production, and to facilitate maintenance and evolution. To accomplish these goals, software engineers have beeing searching for software technologies and methodologies that reduce software complexity, promote reuse, and facilitate evolution [11].

*Separation of concerns* is a software engineering concept used to reduce the complexity of software. It refers "to the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal or purpose" [6]. Concerns help decomposing the software into smaller, more manageable parts. In our days, software's life expectancy increases, so it becomes the subject of major changes which implies the integration, interaction with other pieces of software written by other organizations, and the evolution in new context (new hardware, new operating systems, etc). When software is decomposed into little pieces such as concerns which are well identified and encapsulated, evolution is less costly. This is because changes that are to be made are localized and the impact of change is manageable. Separation of concerns also facilitates reusability and integration. Once concerns are identified, separated and



encapsulated, it must be made possible to integrate, or compose the concerns into other systems [7]. Building software from reusable components will reduce software cost. Integration is also simplified because developers need only concentrate on the relevant relationship among components. Software requirements change fast, so it is very important to have these facilities for software evolution, integration and reusability. Most of the time, it is impossible to predict how software will have to evolve in time. It might require more features, or some of its features have to be changed. Therefore, it is important to be able to decompose and integrate software's features as desired. Multi-Dimensional Separation of Concerns (MDSOC) is one of the AOP approaches which address these issues of software evolution, integration and reusability.

As mentioned previously, concerns help organizing and decomposing the software into manageable and logical parts [7]. MDSOC refers to the separation of concerns involving multiple and arbitrary dimensions. Separation along these dimensions is simultaneously. There is not a dominant dimension of concerns. Most of the present languages and methodologies support separation of concerns toward a single dimension. For example, the object–oriented programming encapsulates data concerns into classes, while other concern types are not effectively represented. This is referred to as *the tyranny of dominant decomposition* [7] because one dominant way of decomposing the software (e.g., by class) makes it difficult to encapsulate other dimensions of concerns. Decomposing the software based on one dimension is considered to be the primary factor in failure of achieving the goals of separation of concerns. Another ability that MDSOC offers is to handle new concerns and new dimension of concerns, dynamically during the software lifecycle. MDSOC allows developers to choose the best modularization scheme, based on the concerns available at the development time. The following are some of the concepts used by MDSOC and therefore by Hyper/J.

- ***Unit***: a syntactic construct in a language (e.g. methods, functions, classes, UML diagrams). In Hyper/J units are member functions, member variables, interfaces, classes and packages.
- ***Concern space***: all units of a software body.
- ***Hyperspace***: a special kind of concern space designed to support MDSOC. The units of a hyperspace are organized in a *multidimensional matrix*. Each *axis* represents a *dimension of concern*. Each *point on an axis* represents a concern in that dimension. The *coordinates of a unit* indicate all the concerns it affects. Figure 2.1 shows the multidimensional matrix. As shown in the pictures, there are two axes (dimensions) shown: Class dimension and Feature dimension. Concerns are grouped into dimensions. Concern Display belongs to Feature (Feature.Display) dimension and Plus (Class.Plus) and Minus (Class.Plus) concerns belong to Class dimension. Concerns in the same dimension are disjoint (i.e., they have no units in common). If concerns in the different dimensions can overlap, concerns in the same dimension cannot overlap.
- ***Hyperplane***: Any single concern within some dimension defines a *hyperplane*. Hyperplane is a set of units that affect a concern. Figure 2.1, shows the

hyperplane defined by the Display Feature, represented by the parallelogram with dashed edges.

- ***Hyperslice***: The hyperspace (or concern matrix) facilitates the identification of concerns, and organizes units according to dimensions and concerns. Hyperslice is a hyperplane that is declaratively complete; it is treated as a module, encapsulating a concern. Declarative completeness means that a hyperslice must declare everything to which it refers. This requirement eliminates coupling between hyperslices, so this will facilitate the integration process. Figure 2.1, shows the shaded area as a hyperslice.

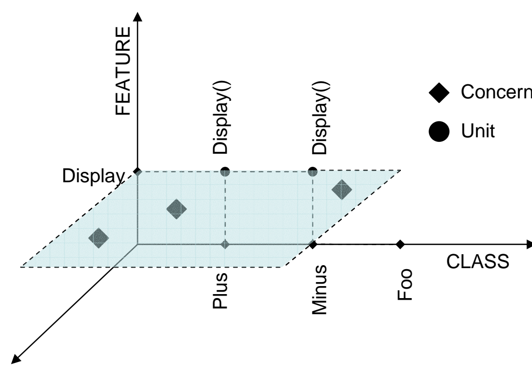



Figure 2.1 Multidimensional matrix Hyperplane defined by Display feature Hyperslice

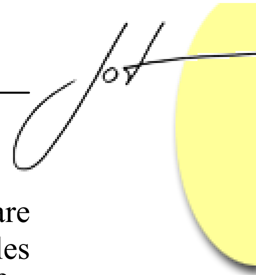
- ***Hypermodule***: a set of hyperslices being integrated and a set of ***integration relationships*** that specify how the hyperslices relate one to another.

## 2.2 Related Works in AOP

AOP implementation on programming languages has been only applied to several languages: Java, C, C++, C#. AspectJ [5], AspectC, AspectC++ [10], and AspectC# [2] are the tools that implements AOP for those languages. Any of these aspect-oriented languages constitutes an extension of the language it comes from, by providing new language constructs to explicitly capture crosscutting concerns (aspect). In any of these languages, aspects are modeled as separate modules that are combined with components (regular classes) at compile-time using a weaver tool.

AOP implementation using MDSOC on programming languages is applied to only two languages so far: Java and VB.NET. Hyper-VB [3] is a tool which adds AOP features to VB.NET by using MDSOC.

Hyper/J is a tool which supports the multi-dimensional separation and integration of concerns in standard Java software [8]. It is an "instance of MDSOC" that enables the definition of Java concerns (hyperslices) and merges the concerns into a complete Java program. With Hyper/J, which has a text based interface, developers build regular Java



classes by choosing the most appropriate decomposition for a program. Java classes are then combined at compile-time by specifying a series of composition rules. Class files produced by Hyper/J are standard Java class files and can be used as input to Hyper/J for further composition. For a full description of the Hyper/J, the reader is referred to the Hyper/J documentation [8]. In this section, a brief overview of the concepts in Hyper/J is given.

The Hyper/J tool permits a set of Java files to be decomposed along multiple dimensions of concern simultaneously. For example, Java classes may be considered as classes in the Class dimension while particular methods may be considered operations in the Feature dimension. Each dimension may be partitioned into a set of concerns.

When using Hyper/J, a developer provides three inputs:

1. A *hyperspace file* that describes the Java class files that can be manipulated by Hyper/J.
2. A *concern mapping file* that describes which pieces of the Java source map to each dimension of concern. In Hyper/J there are five kinds of mapping: package mapping, class and interface mapping, operation mapping, field mapping and "None" concern mapping.
3. A *hypermodule file* that describes which dimensions of concerns should be integrated and how that integration should proceed.

Hyper/J supports the following integration relationships: *mergeByName nonCorrespondingMerge* and *overrideByName*. In *mergeByName* composition strategy, units in different hyperslices with the same name are to correspond. In nonCorrespondingMerge composition strategy, units in different hyperslices with the same name are not to correspond. In overrideByName composition strategy, units with the same name are to correspond, and are to be connected by an override relationship, which causes the last one to override the others in the composed software. The relationship section of every hypermodule specification begins with one of these three strategies. The general strategy may or may not be sufficient to describe the relationship across hyperslices. If it is sufficient, no other relationship will be specified. If it is not sufficient, then Hyper/J supports specializations or exceptions of that strategy for those cases where the strategy is not sufficient. The equate relationship combined with mergeByName or nonCorrespondingMerge is used when units with different names are to be integrated. The bracket relationship indicates that a set of methods should be bracketed, in other words, their invocation should be preceded or followed by other methods.

# 3 THE HYPERC# AOP SYSTEM: FUNCTIONALITY AND SOFTWARE ARCHITECTURE

## 3.1 The HyperC# System Requirements

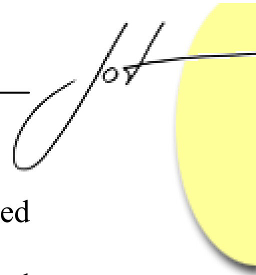
The HyperC# System has the following requirements:

1. The programmers must be able to define classes. The programmers must be able to work on multiple classes in the same time. The programmer must be able to view the previously defined classes while defining the new classes.
2. The programmers must be able to define hypermodules. The programmer must be able to work on multiple hypermodules in the same time. The programmer must be able to view the previously defined hypermodules while defining the new hypermodules.
3. The programmers must be able to compile and run the programs within the system. The compilation result must be displayed to the programmer to indicate the failure or success of the compilation.
4. The system must be able to automatically weave (integrate) concern classes. Programmers must be able to weave (integrate) multiple concern classes at the same time.
5. The system must provide editing capabilities, such as copy, cut and paste.
6. The system must provide different ways of viewing the open windows. Since programmers are allowed to open multiple windows to work on multiple classes and hypermodule, they have to be able to view all the windows they are working on in the same time. To accomplish this, the system offers cascade, a horizontal, and a vertical view for the opened windows.
7. The system must provide a help menu.

## 3.2 The HyperC# Functionality

The following are the functionalities of the HyperC# system.

1. **Source-code pre-processor**. HyperC# System works on source code level. It has a source-code pre-processor, so in order to compile the composed file, uses the C# compiler. One of the important responsibilities of the pre-processor is to perform the integration process. The pre-processor reads in the XML files, performs the integration process and generates a new C# file.
2. **XML representation.** To facilitate the integration process, the HyperC# system generates automatically an XML file for each class or hypermodule defined.
3. **Class editor.** HyperC# System provides a simple GUI-based class editor form. The form has a number of text boxes. They are used by the programmer to input "using" directives, the name of the namespace, global declaration (global



variable), method's name and method's body. The class editor form is discussed with more details in Chapter 4.

4. **Hypermodule editor.** HyperC# System provides a simple GUI-based hypermodule editor form. The form contains a list view box in which is displayed the tree representation of the composition of the defined hypermodule's dimensions. The form contains an input list box for displaying the methods of the loading class. Also, it contains a combo box for defining the dimensions and the concerns contained in those dimensions, and it contains text boxes for defining the *Equate* and *Bracket* relationships. The hypermodule editor form is discussed with more details in Chapter 4.
5. **Class decomposition.** HyperC# System offers a new relationship compared to Hyper/J, the *decompose relationship*. The decompose relationship offers the programmer the possibility to decompose a class based on the concerns defined in the hypermodule. This functionality is available in the hypermodule editor form. In order to decompose a class, the programmer needs to identify and defined the dimensions and concerns and then do the methods mapping. Methods mapping indicates that one or more methods address a given concern in some dimension. Then the programmer has to choose the decompose relationship and build the hypermodule. HyperC# System will create a new class for each concern defined in the hypermodule. The name of the resulting classes will be [class_name]_[concern_name]. For example a Customer class has the following concerns: Identification, ContactInfo, and Purchase. Choosing the decomposed relationship, the HyperC# system will create three new concern classes: Customer_Indentification class, Customer_ContactInfo class and Customer_Purchase class. The Customer class is called the *origin class* and all the resulting classes are *concern classes*.
6. **Three general composition strategies.** Like Hyper/J, HyperC# System supports three general composition strategies. They are called *mergeByName*, *nonCorrespondingMerge*, and *overrideByName*.
7. **Two integration relationships.** HyperC# System supports two integration relationships. These are *Equate relationship* and *Bracket relationship*. Equate relationship has the same purpose as Equate relationship in Hyper/J. There are some differences regarding Equate relationship in Hyper/J and HyperC#. One of the differences is that HyperC# System requires the use of method's signature in Equate relationship, instead of method's name like in Hyper/J. Equate relationship takes an optional *into* specification, which indicates the name that is to be given to the composed method. The other difference is that if no name is specified explicitly, HyperC# will give the name of the first method from the methods to be used in equate relationship. On the other hand, Hyper/J synthesizes a name from the names of the related method. Bracket relationship in HyperC# System indicates the same thing like Bracket relationship in Hyper/J. The same as Equate relationship, Bracket relationship requires the use of method's signature, instead of method's name like in Hyper/J. If in Hyper/J Bracket relationship works on a set of methods, in HyperC# it works on one

method, in other words, the invocation of one method will be preceded and/or followed by other methods.

### 3.2.1 HyperC# Class Decomposition

The main purpose of class decomposition feature is to separate concerns that are tangled in a class. These separated concerns take form of concern classes. HyperC# System makes sure that concern classes are *declaratively complete*, which means that they don't depend on each other. Declarative completeness is important because it eliminates coupling between concern classes. To achieve the declarative completeness, HyperC# System inserts the directives and global declarations of the origin class in each of the concern classes. Concern classes in HyperC# System are the equivalent of hyperslices in Hyper/J. It is the way HyperC# System encapsulates and modularizes concerns.

The concern classes are C# files which have another representation in XML. This saved as XML file is required because it is assumed that those concern classes will be used by the system in the integration process with other concern classes. The integration process requires that all the concern classes to have an XML version of the file. The class which will result after the integration process will be called composed class.

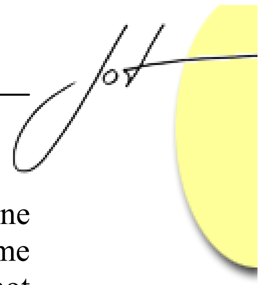
When two or more concern classes from different origin classes are integrated, it is possible to have directives redundant in the composed class, which will cause errors during compilation process. In order to prevent this from happening, HyperC# System has a redundancy check mechanism, which will guarantee the single appearance of directives.

Other kinds of redundancy that can happen are private variable redundancy and global variable redundancy. Private variable redundancy can not be avoided, because the programmer is unable to predict which classes will be decomposed (based on which concerns) and how they will be integrated further. This kind of redundancy will cause errors during compilation. When this happens, the programmer will change the name of the variables manually. For global variable redundancy, the HyperC# System has a redundancy check mechanism, which will also guarantee the single appearance of global variables.

### 3.2.2 HyperC# Class Integration

The class integration feature is used to integrate concerns in the form of concern classes, in order to create new software that satisfies particularly new requirements. HyperC# System uses the same integration method as Hyper/J with some minor modifications.

One of the differences between HyperC# and Hyper/J is regarding "None" concern. HyperC# System creates and add a "None" concern anytime the programmer creates a new dimension. Hyper/J automatically puts any methods not assigned to any other concern in the dimension into the "None" concern. When a method from a concern is calling a method from "None" or any other concern, Hyper/J adds the abstract declaration of the method which is called in the calling method (this is how declarative completeness



is achieved in Hyper/J). HyperC# System does not have the capability to determine if one method from one concern is calling another method from another concern under the same dimension. In case one method from one concern is calling a method which does not belong to any concern under the same dimension, the called method should be put it under the "None" concern. This means, in case a method is calling one of the "None" methods, the system will add the complete body of the method in the composed class. If there is a method class that does not belong to any of the concerns under the same dimension, and that method is not called by any of the concern's method, that method will not be added to the "None" concern. This is HyperC#'s simplification, compared to Hyper/J System.

### 3.3 HyperC# System Components

Figure 3.1 depicts the software components of the HyperC# System, with an arrow indicating that a component calls another. The software components are the following:

- **Central System Control with GUI**. Call various components based on programmer input; captures the output and passes it to the appropriate components; GUI captures te programmer input and presents output to the programmer.
- **Hyper C# Pre-processor**. Reads in the XML files, performs the weaving process
- and generates a new C# file.
- **XML Manager**. Generates XML, and modifies XML.
- **C# Compiler**. Microsoft's existing C# compiler is used in the system.

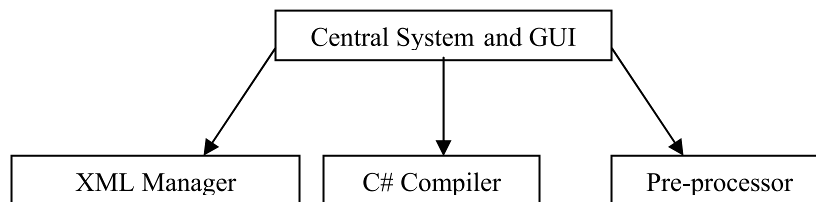



Figure 3.1 High –level system software architecture

## 4 THE HYPERC# AOP SYSTEM: DESIGN AND IMPLEMENTATION

HyperC# System provides tools for C# programmers to use MDSC features. The HyperC# System was developed using C# language. C# provides a wide set of library that was used to create the graphical user interface of the system. In this section the system's graphical interface will be presented.

### 4.1 The Classes and the GUI's Forms of the System

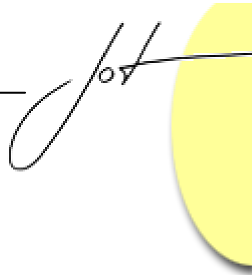
The system is composed of eight classes. Seven of these classes are forms that HyperC# programmers can interact with. The other class is an auxiliary class used by one of the

classes. Those classes are MainMenuForm, InputForm, ClassForm, SourceCodeAndTreeView, HyperForm, CompileAndRunClasses, HelpForm, and XMLto_SourceCode. The first seven classes are windows form with numbers of controls residing on. The last class is the helping class which has the methods declared as public, so any other class can access them. Each class has the following functionalities:

- The class MainMenuForm is the system start point. When the system starts up this form is shown. This form is the container for the other forms (InputForm, ClassForm, SourceCodeAndTreeView, CompileAndRunClasses and HyperForm, HelpForm). This form contains menus that programmers can choose to perform actions which they desire. For example,a programmer may want to create a new class or a new hypermodule.
- The class InputForm is used by the system when programmers want to create a new class or a new hypermodule. It prompts the programmer for the name of the new created class or hypermodule.
- The class ClassForm is used to create new classes. The class contains label, text box and buttons to facilitate programmer to enter the data and the system to create the new class.
- The class SourceCodeAndTreeView is used to view classes and hypermodules. The class contains the text box and tree view objects. The text box is used to display the source code in a text format. The tree view is used to display the source code in a tree format.
- The class HyperForm is used to create new hypermodules. The class contains label, text box, tree view, list box objects to facilitate programmers to enter data and the system to create the new hypermodule.
- The class CompileAndRunClasses programmatically calls the C# compiler to compile the files and run the executable file.
- The class HelpForm is the help menu for HyperC#. When clicking on it, a help form is opened. The system reads the contents of a text file which resides where the program resides and writes the contents of the file in the text box of the form.
- The class XMLto_SourceCode: This class does not have a form. It contains only helping methods that are used by other classes.

### 4.1.1 A “Main Menu” Form

This form is the programmer start point. The logical structure of the form can be seen in Figure 4.1. Figure 4.2 is the form to create a new class.



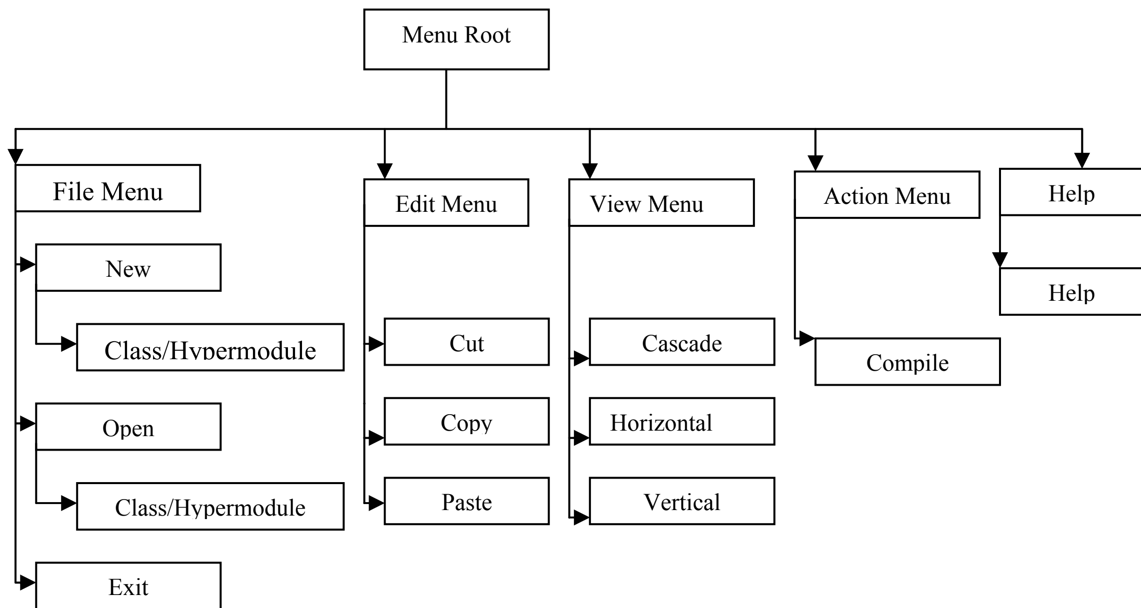



Figure 4.1 The main menu hierarchy

### 4.1.2 An “InputForm” Form

This form is used for prompting the programmer for a class name or hypermodule name. This form is opened by the system when the programmer clicks File → New → Class or File → New → Hypermodule. The form contains a text box where the programmer has to enter the name of the new class or the new hypermodule he/she wants to create. The form contains an “OK” buttons that after the programmer clicks it, a new form is open.

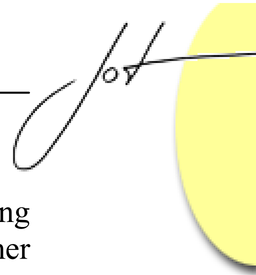
### 4.1.3 A “Create a New Class” Form

This form contains various controls that programmers can use to define a new class. Text boxes are used to enter the source code. Buttons are used to indicate when the programmer has done entering the code. This form will create only one class at a time. If the programmer wants to create more than one class, he/she has to open multiple forms.

The system form gives the programmer the possibility to change the class in case he/she wants to add a new method, modify a method, delete a method, and change the declarations or the directives. At the end of the process, there will be two files created for each class created, a text version and an XML version. Figure 4.2 shows the interface of the form.

Figure 4.2 A form to create a new class

1. "Directives" label: Clicking the "Add" button for the "using" directive causes the system to write the directive's names to an XML file. To avoid changes until the XML file is created, the button is disabled after the click.
2. "Namespace" label: Clicking the "Add" button for a namespace causes the system to write the namespace to an XML file. Since a class belongs to only one namespace, the button is disabled after the click.
3. "Members" label: Clicking the "Add" button for the class members causes the system to write all the members to an XML file. To avoid changes until the XML file is created, the button is disabled after the click.
4. "Add Method" button: Clicking this button causes the system to write methods names and methods bodies to an XML file. Multiples methods can be added.
5. "Commit" button: Clicking this button, the system will create the corresponding XML file.
6. "Directives" radio button: Clicking this button, the "Add" button for adding "using" directives, will be enabled. The button being enabled, will allow the programmer to change the directives (to add or to delete) that the class needs.



7. “Members” radio button: Clicking this button, the “Add” button for adding members, will be enabled. The button being enabled, will allow the programmer to change the members (to add or delete) that the class needs to declare.
8. “Method” radio button: Clicking this button, the “Add New Method”, ”Modified Method” and “Delete Method” buttons are enabled.
9. “Add New Method”: Clicking this button, the system will add a new method to the XML file. All the changes are saved in the XML file.
10. “Modify Method”: Clicking this button, the programmer will be able to modify a method’s body. When the programmer selects the method from the combo list to be modified, the body of the method will be displayed in the text box. The programmer will make the necessary changes in the text box. After clicking the “Modify Method” button, the system will read the XML file and search for the name of the method the programmer chose and write back the new body of the method. All the changes are then saved in the XML file.
11. “Delete Method”: Clicking this button, the programmer will be able to delete a method that the class does not need it anymore. The programmer has to select from the combo list the method to be deleted. After clicking the “Delete Method” button, the system will read the XML file and search for the name of the method and delete the nodes of method’s name and method’s body from the XML file. All the changes are then saved in the XML file.
12. “Add Class” button: Clicking this button, the system will strips out the tags of the XML file and writes the content of the XML file to a text file. In this way, the text version file of the class will be created. At this point, there are two files created for the new class created, the text version and the XML version. Since the class has already been created with both of the versions of files, all the buttons will be disabled after this click.
13. “Cancel” button: The programmer clicks this button to cancel the class just created. The system will delete the text version and the XML version of the class.
14. “Close” button: The programmer clicks this button to close the form.

### 4.1.4 A “Hypermodule declaration” Form

This form contains various controls that programmers can use to create a new hypermodule file. The hypermodule will have also three versions of files, a text version and two XML versions.

The form provides a tree view of the hypermodule’s dimensions. The “None” concern is the default concern in a dimension. Every time a new dimension is created, a “None” concern is automatically added by the system. First, all the dimensions needed should be created. After each dimension, the programmer has to input the corresponding concerns. The next step is to load methods from different classes and map them with the corresponding concern under the specific dimension. Figure 4.3 shows the interface of the form.

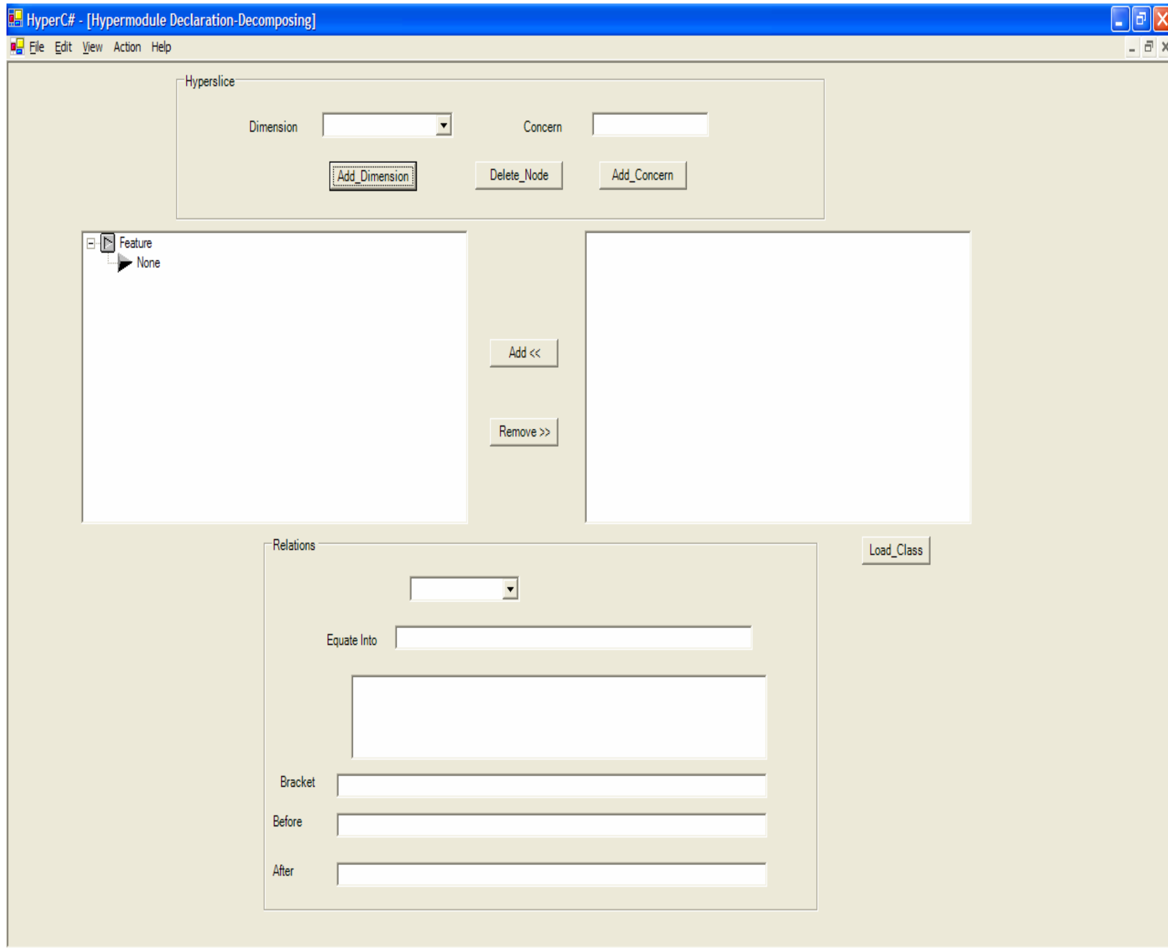



Figure 4.3 A form to create a new hypermodule

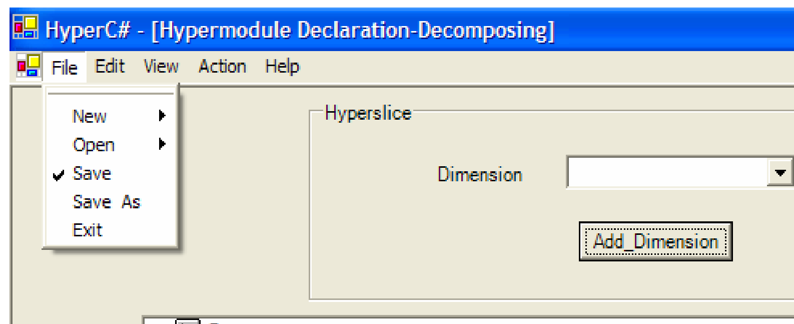
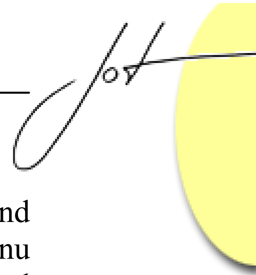



Figure 4.4 The "File" menu contents of the hypermodule form

1. "File"menu: It contains menu items as seen in Figure 4.4. The "New" and "Open" menus belong to the Main Menu form. When the hypermodule form is maximized (this is the default window state), the menu from the "Main Menu" Form and this form will merge into this hypermodule menu. The menus that



belong only to the “Hypermodule Declaration” form are “Save”, “Save As” and “Exit”. “Save” menu is used to save the current hypermodule. “Save As” menu is used to save current hypermodule into a different name. Using “Save” and “Save As” menus, the system will save the hypermodule file in an XML format. “Exit” menu is used to close the current form.

2. ”Action” menu: “Compile” menu belongs to the Main Menu form. “Build” menu belongs to “Hypermodule Declaration” form and is used to build the hypermodule. Figure 4.5 shows the contents of the “Action”menu. When the hypermodule form is maximized (this is the default window state), the menu from the “Main Menu” Form and this form will merge into hypermodule menu shown in Figure 4.5.

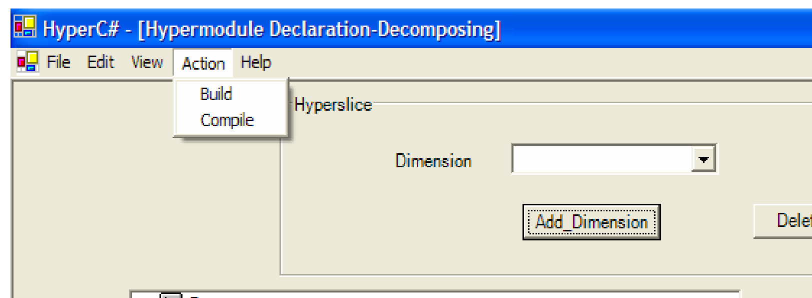
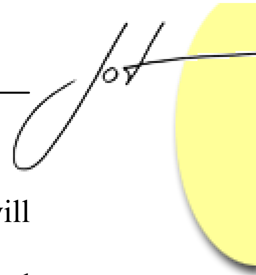



Figure 4.5 The “Action” menu of the hypermodule form

3. “Edit” menu: This menu belongs to the “Main Menu” form. It is merged with hypermodule menu when the “Hypermodule Declaration” form is maximized. Because it belongs to the “Main Menu” form it has the same content as the “Main Menu” form’s menu, the copy, cut and past selected text from the MDI child forms.
4. “View” menu: The same as “Edit” menu, it belongs to the “Main Menu” form and it is merged with the “Hypermodule declaration” form’s menu.
5. “Help” menu: It belongs to the “Main Menu” form and it is merged with the “Hypermodule declaration” form’s menu.
6. Dimension composition tree view. This tree view displays the current dimensions composition, including the concerns that are in each dimension.
7. Dimension Combo Box. When the programmer wants to create a new concern, he/she has to choose a dimension from the combo box where the concern has to belong to.
8. Concern Input Box. This box is used by the programmer to enter the name of the concern he/she wants to create.
9. “Add_Dimension” button: This button is used to create a new dimension. When this button is clicked, the new dimension is added to the Dimension Combo Box and a new node with the name of the new dimension will be added to the Dimension composition tree view. Every time a new dimension is created, the system will automatically adds a “None” concern, as the first concern under the specified new dimension.

10. "Add_Concern" button: This button is used to add a concern stated in the concern input box into the dimension shown on the dimension combo box. In the same time, a new subnode with the name of the concern will be added in the Dimension composition tree view under the dimension selected from the dimension combo box.
11. "Delete_Node" button: This button is used to delete any dimension, concern, or method. One has to select the dimension, the concern or the method to be deleted by clicking on it in the tree view and then to click this button.
12. "Add << " button: This button is used to move methods from the loaded-class list box to dimension composition tree view. First, the programmer has to select a concern by clicking on it in the tree view, choose a method from the loaded-class list box by clicking on it, and then click on the "Add <<" button.
13. "Remove >>" button: This button will remove the method from the dimension composition tree view to the loaded-class list box. First, the programmer has to select the method by clicking on it in the tree view and then click the "Remove >>" button. If the currently loaded class is not the one that the methods belong to, the system will generate an error.
14. Loaded class name: This label displays the name of the class that is displayed in the loaded-class list box.
15. Loaded-class list box: This list box displays only the methods' signatures of the loaded class.
16. Load_Class button: This button is used to load the methods of a class into the loaded-class list box, so the programmer will be able to map the necessary methods into the corresponding concerns under corresponding dimensions.
17. Relationship Combo Box: This combo box contains all the relationship available in the system: mergeByName, nonCorrespondingMerge, overrideByName, and decompose. The programmer will have to choose one of them when creating the hypermodule.
18. "Equate Into" text box. This text box contains the name of the composed method that will result after using "equate" relationship. If no name is specified in the "Equate Into" text box, HyperC# will use the name of the first method from the set of methods specified in the equate relationship, as the name of the composed method.
19. "Equate" list box: This list box will display all the methods used in "equate" relationship. In order to include a method in this list box, the programmer has to select a method from the dimension composition tree and drag and drop it into the "Equate" list box. If the programmer wants to remove one or more of the methods from the "Equate" list box, he/she has to select the methods that have to be removed and then right-click on the methods. This will trigger the appearance of a context menu with two options: Remove and Clear All. If the programmer wants to delete only the selected methods, than he/she has to choose the "Remove" option. If the programmer wants to delete all the methods from the "Equate" list box, the "Clear All" option has to be chosen.



20. “Bracket” text box: This text box will display the name of the method that will bracket.
21. “Before” text box: This text box will display the method that will be called before the method specified in the “Bracket” text box. In order to put the name of the method in this text box, the programmer has to load the class which contains this method into the loaded-class list box. Then, the programmer has to select the appropriate method and right click on the “Before” text box. A context menu with two options, “Before” and “After”, will pop up. Choosing “Before” option will make the name of selected method to be displayed into the “Before” text box.
22. “After” text box: This text box will display the method that will be called after the method specified in the “Bracket” text box. In order to put the name of the method in this text box, the programmer has to load the class which contains this method into the loaded-class list box. Then, the programmer has to select the appropriate method and right click on the “After” text box. A context menu with two options, “Before” and “After”, will pop up. Choosing the “After” option will make the name of selected method to be displayed into the “After” text box.

### 4.1.5 A “View Class/Hypermodule in Text and Tree Mode” Form

This form is used by the programmers to see the previously defined classes and hypermodules. For each class and hypermodule that are created using this system, its XML version of the file is also created. Using this form, the programmer can see the source code in either text format or XML format. In XML format, the system creates a tree and represents each XML tag as a tree node.

While developing more classes or hypermodules, the programmer can open multiple forms .The system provides three viewing layouts: cascade, horizontally, and vertical.

### 4.1.6 A “Compile and Run” Form

To compile classes, programmers have to select valid files from the directory list. A valid file has the extension “.cs”. The form contains a tree view which contains the list of all files from the directory specified in the text box. When the form loads, the tree view contains the list of all files from the current directory. If the programmer wants to change the directory, then he/she has to change the name of the directory in the text box. The form contains a list box in which the programmer will move the files to be compiled. After the files are chosen, clicking the “Compile” button, the system will call the C# compiler and prints the result of the compilation on the text box. Upon a successful compilation, the programmer clicks the “Run” button to execute the executable file created after compilation. If the compilation failed, programmers must correct the error to be able to proceed.

# 5 REAL WORLD APPLICATIONS

In this section the use of HyperC# in real world is discussed. The examples in this section are about a Wholesale Corporation. In order to give its customers rebates, the company has a Customer Evaluator Software which evaluates each customer based on how many items each customer bought. The company's management wants to be more specific and have separate evaluator software for resident customers, such as Resident Customer Evaluator Software. The new evaluator's software will print for each resident customer the number of items bought and also the amount of money spent in one month. The company will use HyperC# to satisfy the requirements. They want to keep the old evaluation mechanism (based on number of items bought) while introducing the new mechanism. In order to accomplish their task, the company will use mergeByName relationship to merge the old evaluation mechanism (which requires only outputting the number of items bought) with the new one (which requires outputting the number of items and the amount of money spent). The evaluator software uses a Microsoft Access database and a class Customer.

## 5.1 Relational Database Schema

In the old evaluation application, Customer class is using a database schema. This database schema is presented in Figure 5.1. To accommodate the requirements changes, it is necessary that the old database schema to be extended as shown in Figure 5.2

**CUSTOMER**

| ConsID | Name | Address | SSN | TellNr | ShipAddress | Email |
|---|---|---|---|---|---|---|

**PURCHASE**

| ConsID | MoneySpent | NrItems |
|---|---|---|

**Figure 5.1 The original evaluator's database schema**

**CUSTOMER**

| ConsID | Name | Address | SSN | TellNr | ShipAddress | Email |
|---|---|---|---|---|---|---|

**PURCHASE**

| ConsID | MoneySpent | NrItems |
|---|---|---|

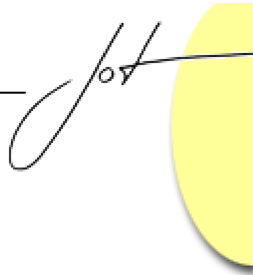
**BUSINESS_PURCHASE**

| ConsID | BusinessMoney |
|---|---|

**RESIDENT_PURCHASE**

| ConsID | PurchaseTime | ResidentMoney |
|---|---|---|

Figure 5.2 Extended evaluator's database schema



## 5.2 Customer Class

The Customer class contains the methods shown in Figure 5.3 A. Because the Customer class addresses more than one concern, it is necessary to decompose the original class. The Customer class addresses three concerns: Identification concern, Contact_Info concern, and Purchase concern. Initially the evaluation software does not need any of the methods related to the Contact_Info concern, so there is no need to include them in the new software.

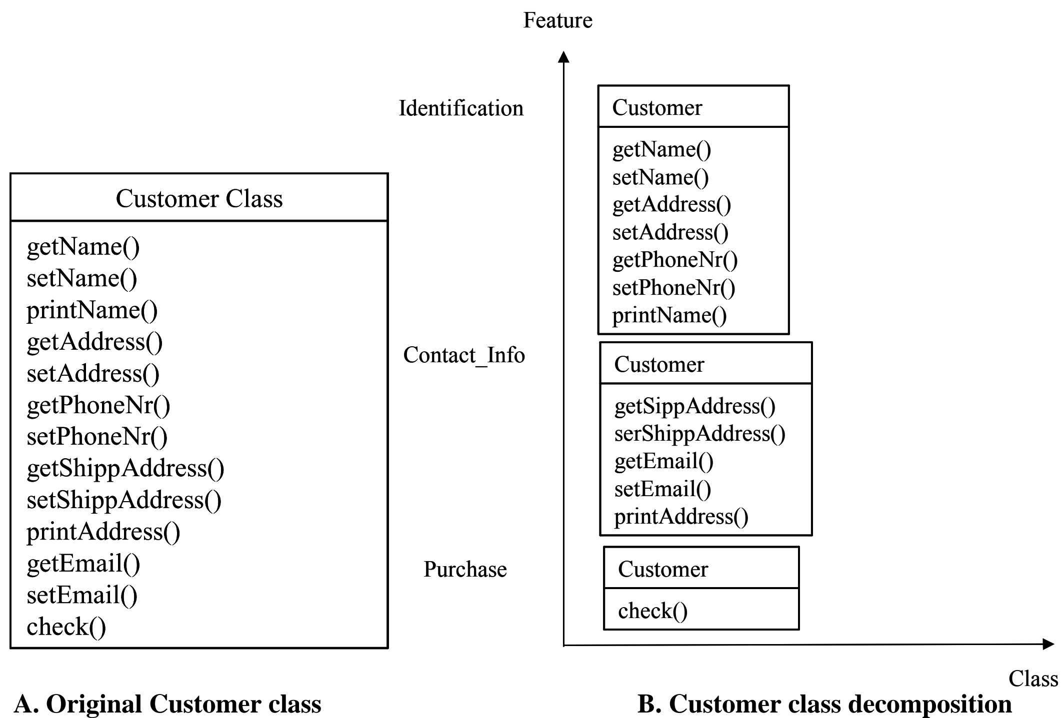



Figure 5.3 Customer class

Figure 5.3 B shows how Customer class can be decomposed based on the concerns defined. In order to accomplish this class decomposition, first the programmer will have to use the HyperC# class editor to create the Customer class. After that, the next step is to use Hypermodule editor with decompose relationship chosen. First, a new hypermodule file is created. The programmer will have to identify and create the dimensions and the corresponding concerns. The dimension created is Feature and the corresponding concerns are None (added by the HyperC# System automatically), Identification, Contact_Info and Purchase. Then, the Customer class is loaded into the editor by clicking "Load_Class" button. Figure 5.4 presents the hypermodule editor with dimensions and concern created and Customer class loaded.

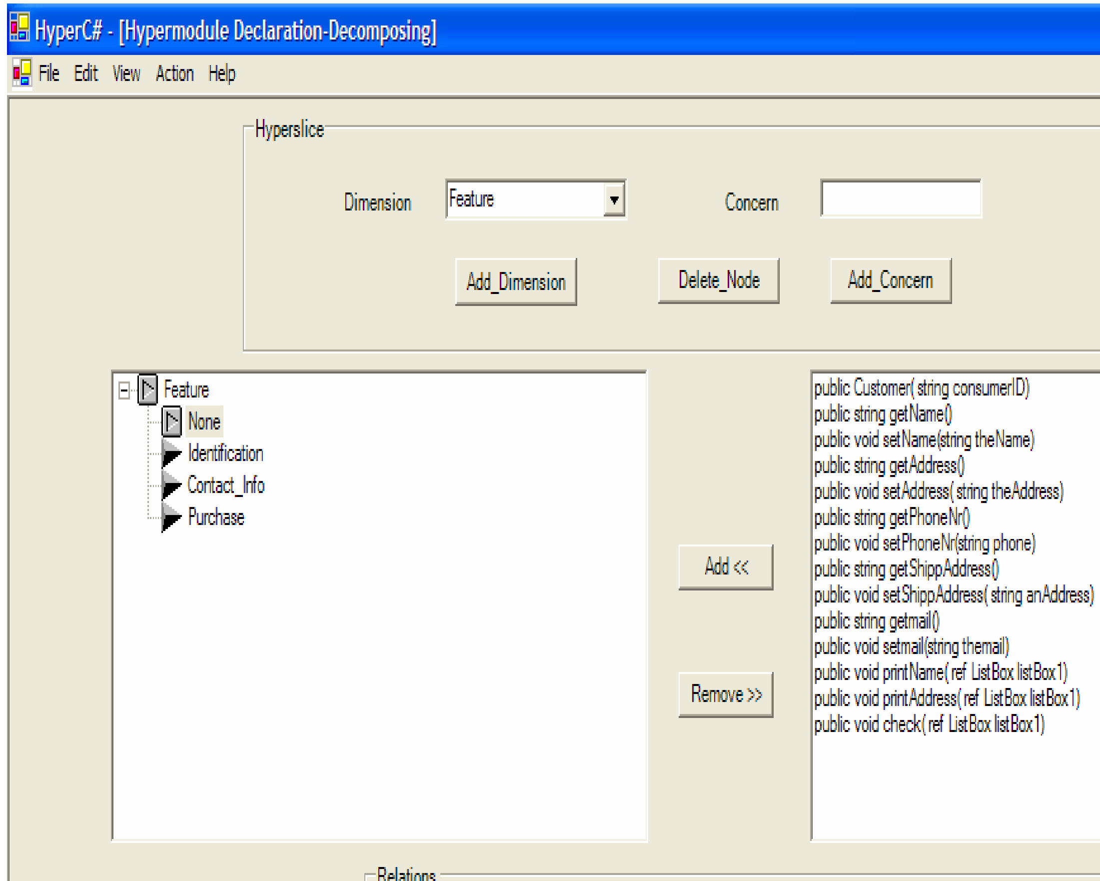
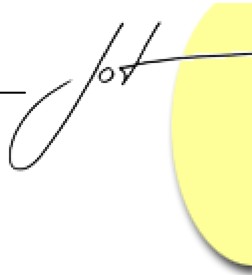



Figure 5.4 Hypermodule editor with Customer class loaded

After this, follows the mapping of methods to the corresponding concerns. As can be seen in Figure 5.5, the constructor of the Customer class gets mapped to the None concern, the functions to retrieve and change the name of the customer, the function to retrieve and change the customer's phone number, the function to retrieve and change the address of the customer are mapped to Identification concern. The functions to retrieve and change the shipping and e-mail addresses are mapped to Contact_Info concern. The function to print the number of items bought by a customer is mapped to Purchase concern.



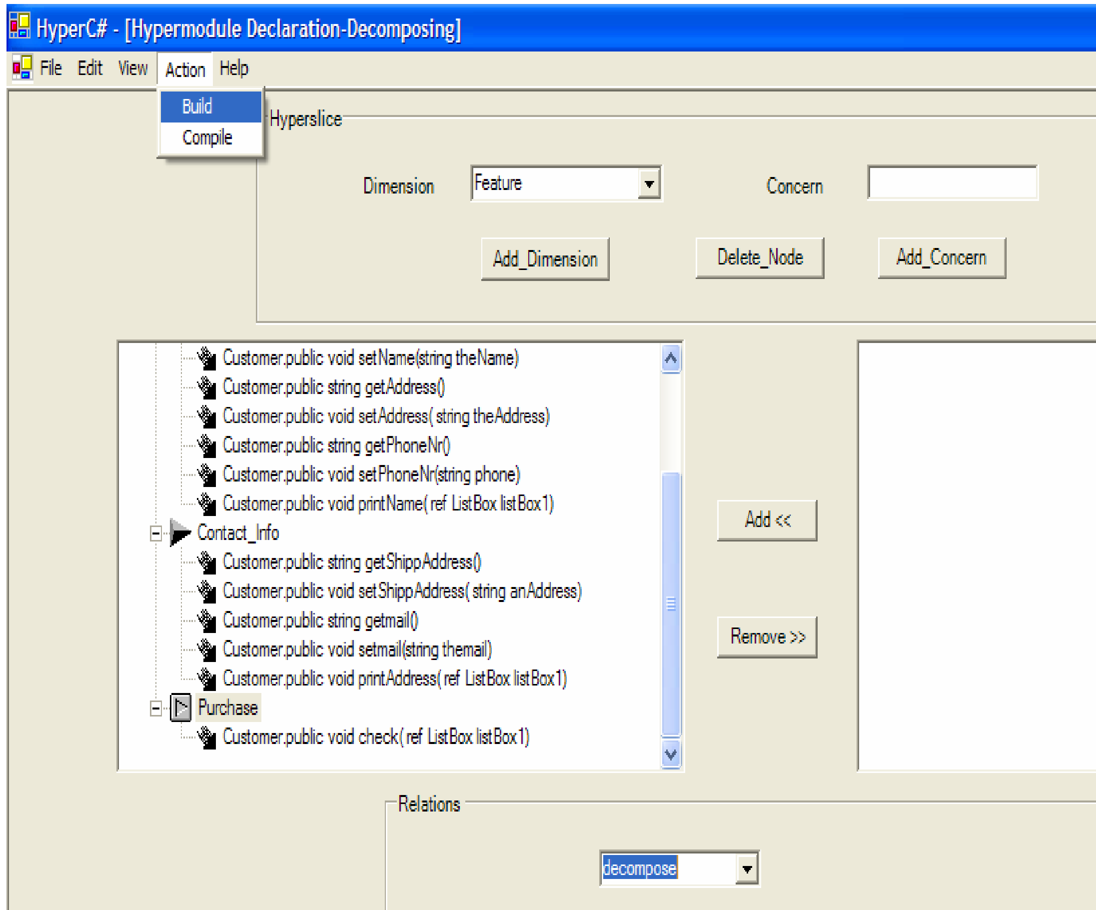



Figure 5.5 Hypermodule editor with mapping of Customer class

After choosing *decompose* relationship and clicking on the Build menu, HyperC# System creates three concern classes: Customer_Identification class, Customer_Contact_Info class and Customer_Purchase class.

## 5.3 Creating New Classes

In order to perform the required changes to the software,a new classe has to be created. This new classe is: Customer_ResidentPurchase class. Figure 5.6 presents the Customer_ResidentPurchase class and its check () method.

HyperC# - [SourceCode-Or-TreeView-C:\BegCSharp\Proj_2_22_05\Master_project\bin\Debug\Customer_Reside

File Edit View Action Help

Source Code View | Source Code Tree View

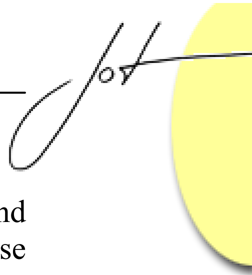
```
private OleDbDataAdapter dAdapter;
private DataSet dSet;

public Customer_ResidentPurchase( string consumerID)
{
try
  {
          consID = consumerID;
          con = new OleDbConnection(conString);
          con.Open();
          dAdapter=new OleDbDataAdapter("SELECT * FROM Customer WHERE ConsID = consID ", con);
          dSet=new DataSet();
          dAdapter.Fill(dSet,"Customer");
          name = dSet.Tables["Customer"].Rows[0]["Name"].ToString();
          address = dSet.Tables["Customer"].Rows[0]["Address"].ToString();
          ssn = dSet.Tables["Customer"].Rows[0]["SSN"].ToString();
          telnr = dSet.Tables["Customer"].Rows[0]["TelNr"].ToString();
          shipingaddress = dSet.Tables["Customer"].Rows[0]["ShipAddress"].ToString();
          email = dSet.Tables["Customer"].Rows[0]["Email"].ToString();
          con.Close();
  }
  catch(Exception ex)
  {

          Console.WriteLine("Error : " + ex.Message);

  }
}

public void check(ref ListBox listBox1)
{
string temp1;
con = new OleDbConnection(conString);
con.Open();
dAdapter=new OleDbDataAdapter("SELECT * FROM ResidentPurchase WHERE ConsID = consID ", con);
dSet=new DataSet();
dAdapter.Fill(dSet,"ResidentPurchase");
DataColumn[] keys1 = new DataColumn[1];
keys1[0] = dSet.Tables["ResidentPurchase"].Columns["ConsID"];
dSet.Tables["ResidentPurchase"].PrimaryKey = keys1;
DataRow findRow1 = dSet.Tables["REsidentPurchase"].Rows.Find(consID);
temp1 = findRow1["ResidentMoney"].ToString();
Console.WriteLine("Money spent:  {0} ", temp1 );
listBox1.Items.Add("Money spent: " + temp1);
con.Close();
}

}//end of the class
}//end of the namespace
```

start | 3 Windows Expl... | 2 Notepad | Master_project - ... | HyperC# - [Sourc...

Figure 5.6 Customer_ResidentPurchase class and its check() method

## 5.4 Integration Process

In order to create the new evaluator software for resident customers (the one to print the number of items bought and the amount of money spent), a new customer class has to be created. To achieve this, some concern classes have to be integrated. The software developer identifies two concerns: Identification concern and Purchase concern. The next step is the mapping process when methods from different concern classes have to be mapped to the corresponding concern. The concern classes that are integrated are:



Customer_Identification class, Customer_Purchase class, and Customer_ResidentPurchase class. Figure 5.7 presents the ResidentPurchase hypermodule form which will accomplish the integration process.

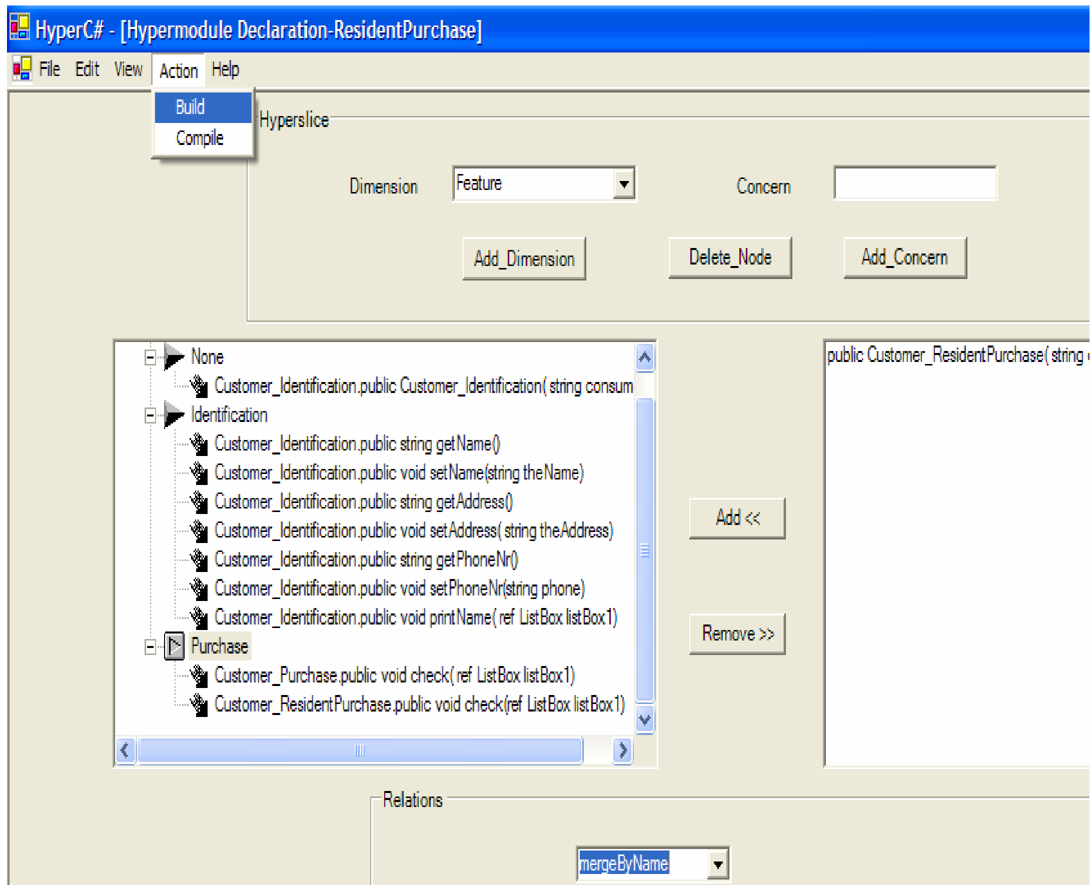



Figure 5.7 ResidentPurchase hypermodule using mergeByName relationship

As can be seen, under Identification concern, the developer puts all the methods from Customer_Identification class. Under Purchase concern, the developer puts check () methods from Customer_Purchase class and Customer_ResidentPurchase. After building the ResidentPurchase hypermodule, a new Customer class will be created that accommodates the new requirements for resident customers. The customer evaluation will be based on the number of items bought in one month (which was the original customer evaluation) and based on the money spent in one month (which was added as a new requirement). The check () method from Customer_Purchase class will be merged with check () method from Customer_ResidentPurchase class. As can be seen, in order to create the new software, there was no need to write more code, it was just a matter of integrating pieces of code together.

Figure 5.8 shows how these check () methods are integrated to form a new check() methods in the ResidentPurchase hypermodule. As can be seen, the bodies of these two methods are merged under a new check () method.

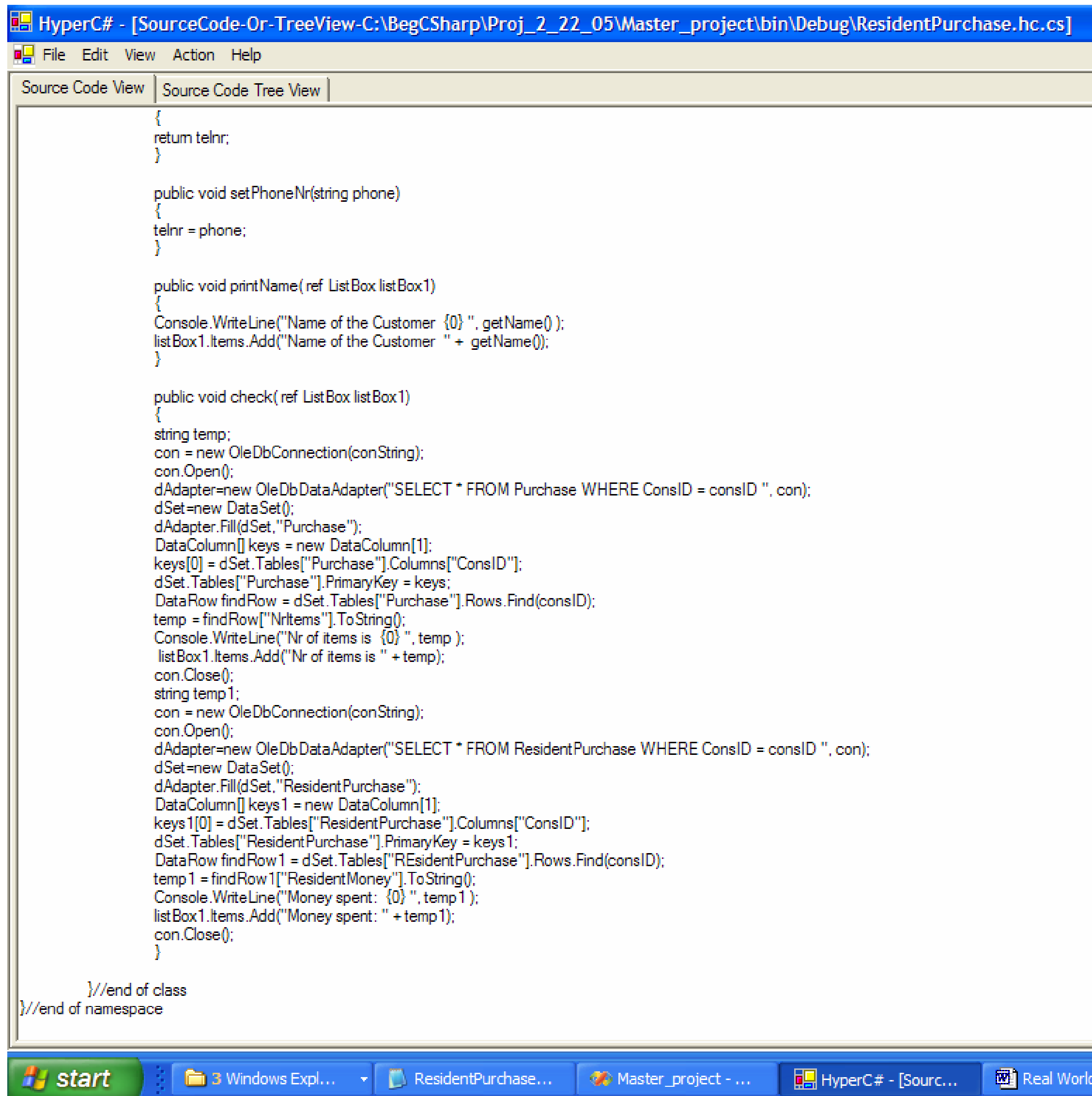
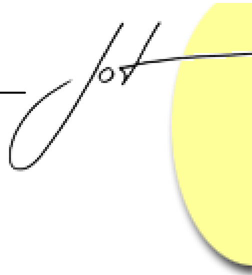



Figure 5.8 ResidentPurchase hypermodule code file

Figure 5.9 shows the screenshot of the original customer evaluation. Figure 5.10 shows the screenshot of the Resident Customer evaluation.



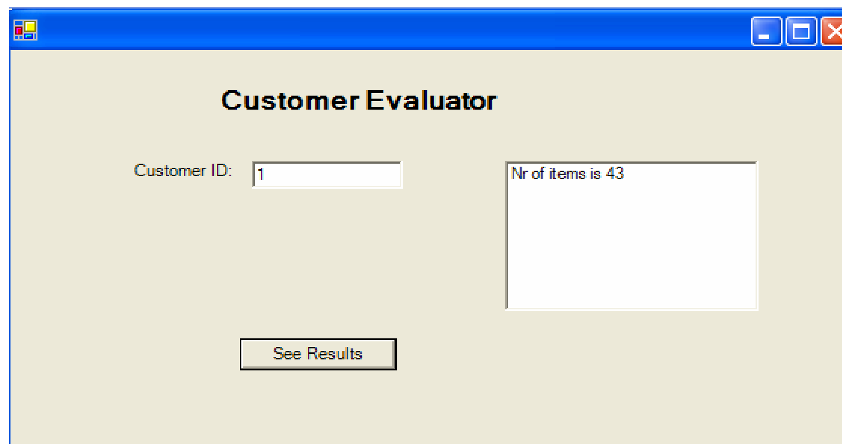



Figure 5.9 Original customer evaluator

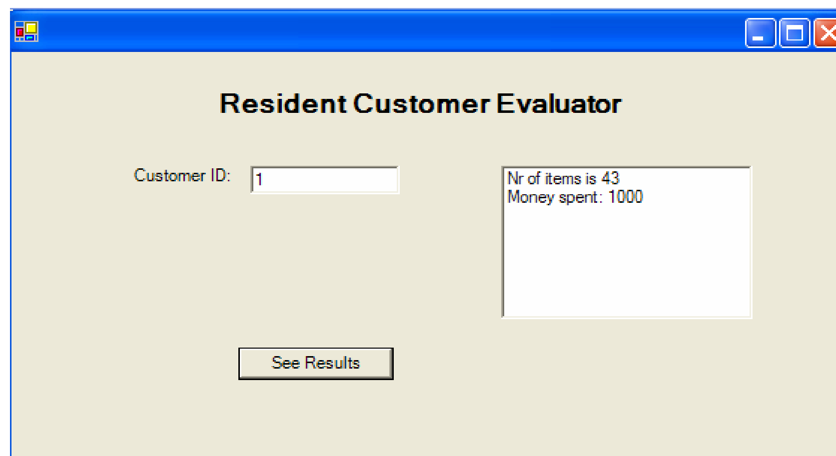



Figure 5.10 Resident Customer evaluator

## 5.5 Discount Evaluation

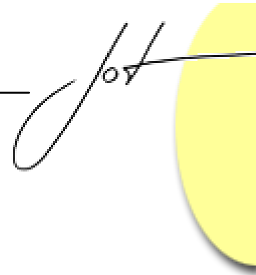
The company wants to make some discounts to its resident customers. In order to do this, the company has to create a program which will display the discount of each resident customer. The discount will be based on the money the customer spent in one month and based on the number of times the customer bought products from the store during the same month. The software will display all of these for each resident customer. These are the new requirements for the software. The customer evaluator needs to evolve, so it can evaluate the discount of each resident customer. In order to accomplish this, two more concern classes will have to be created. They are Customer_ResidentDiscountOnMoney class and Customer_ResidentDiscountOnNrPurchases class. Customer_ResidentDiscountOnMoney class contains a method discount_on_money() which will calculate based on the data obtained from the store's database, the discount for

money spent by the resident customer during one month. Figure 5.11 presents the created new class and the new method for calculating the discount.

```
HyperC# - [SourceCode-Or-TreeView-C:\BegCSharp\Proj_2_22_05\Master_project\bin\Debug\Customer_ResidentDiscountOnMoney.cs]
File  Edit  View  Action  Help
Source Code View | Source Code Tree View |

            ssn = dSet.Tables["Customer"].Rows[0]["SSN"].ToString();
            telnr = dSet.Tables["Customer"].Rows[0]["TelNr"].ToString();
            shipingaddress = dSet.Tables["Customer"].Rows[0]["ShipAddress"].ToString();
            email = dSet.Tables["Customer"].Rows[0]["Email"].ToString();
            con.Close();
   }
   catch(Exception ex)
   {
            Console.WriteLine("Error : " + ex.Message);

   }
}

public void discount_on_money(ref ListBox listBox1)
{
try
    {
      string temp1;
      con = new OleDbConnection(conString);
      con.Open();
      dAdapter=new OleDbDataAdapter("SELECT * FROM ResidentPurchase WHERE ConsID = consID ", con);
      dSet=new DataSet();
      dAdapter.Fill(dSet,"ResidentPurchase");
      DataColumn[] keyss = new DataColumn[1];
      keyss[0] = dSet.Tables["ResidentPurchase"].Columns["ConsID"];
      dSet.Tables["ResidentPurchase"].PrimaryKey = keyss;
      DataRow findRow1 = dSet.Tables["ResidentPurchase"].Rows.Find(consID);
      temp1 = findRow1["ResidentMoney"].ToString();
      con.Close();

            if(Convert.ToInt32(temp1) < 1000 && Convert.ToInt32(temp1) > 200)
            {
              Console.WriteLine("Discount on money : 10 % ");
              listBox1.Items.Add("Discount on money : 10 % ");
            }
            else if (Convert.ToInt32(temp1) >=1000)
            {
              Console.WriteLine("Discount on money: 15 %");
              listBox1.Items.Add("Discount on money : 15 % ");
            }


   }
   catch(Exception ex)
   {
      Console.WriteLine("Error : " + ex.Message);
   }
}

}//end of the class
}//end of the namespace

start | Project_Docume... | Debug | Project_examples | BusinessPurchas... | Master_project -... | Real W
```

Figure 5.11 Customer_ResidentDiscountOnMoney class and discount_on_money method

Customer_ResidentDiscountOnNrPurchases class contains a method discount_on_nr_purchases() which will calculate, based on data obtained from the store's database, the discount for the number of times the resident customer made purchases during one month. Figure 5.12 presents the created new class and new method for calculating the discount.



HyperC# - [SourceCode-Or-TreeView-C:\BegCSharp\Proj_2_22_05\Master_project\bin\Debug\Customer_ResidentDiscountOnNrPurchases.cs]

File Edit View Action Help

Source Code View | Source Code Tree View

```
            ssn = dSet.Tables["Customer"].Rows[0]["SSN"].ToString();
            telnr = dSet.Tables["Customer"].Rows[0]["TelNr"].ToString();
            shipingaddress = dSet.Tables["Customer"].Rows[0]["ShipAddress"].ToString();
            email = dSet.Tables["Customer"].Rows[0]["Email"].ToString();
            con.Close();
  }
  catch(Exception ex)
  {
            Console.WriteLine("Error : " + ex.Message);

  }
}

public void discount_on_nr_purchases(ref ListBox listBox1)
{
try
  {
     string temp2 ;
     con = new OleDbConnection(conString);
     con.Open();
     dAdapter=new OleDbDataAdapter("SELECT * FROM ResidentPurchase WHERE ConsID = consID ", con);
     dSet=new DataSet();
     dAdapter.Fill(dSet,"ResidentPurchase");
     DataColumn[] keys2 = new DataColumn[1];
     keys2[0] = dSet.Tables["ResidentPurchase"].Columns["ConsID"];
     dSet.Tables["ResidentPurchase"].PrimaryKey = keys2;
     DataRow findRow2 = dSet.Tables["ResidentPurchase"].Rows.Find(consID);
     temp2 = findRow2["PurchaseTime"].ToString();
     con.Close();

            if(Convert.ToInt32(temp2) < 5 && Convert.ToInt32(temp2) > 25)
            {
              Console.WriteLine("Discount on nr purchases : 1% ");
              System.Console.In.Read();
              listBox1.Items.Add("Discount on nr purchases : 1% ");
            }
            else if (Convert.ToInt32(temp2) >=25)
            {
              Console.WriteLine("Discount on nr purchases : 2% ");
              System.Console.In.Read();
              listBox1.Items.Add("Discount on nr purchases : 2% ");
            }
  }
  catch(Exception ex)
  {
            Console.WriteLine("Error : " + ex.Message);
  }
}

}//end of the class
}//end of the namespace
```

start | Project_Docume... | Debug | Project_examples | BusinessPurchas... | Master_project -... | Real World App

Figure 5.12 Customer_ResidentDiscountOnPurchases class and discount_on_nr_purchases() method

In order to satisfy the requirements, Equate relationship is used in combination with the mergeByName general relationship. The requirements state that the discount evaluator print the name of the customer and mailing address before printing the discounts, so to know to whom to send the discounts. For printing the personal information of the resident customer (name and mailing address), there is the need of bracket relationship, which is used to bracket discount_on_money() method with the method printAddress() from Customer_Contact_Info concern class. Method printAddress() is called before discount_on_money() method is called. This will make possible to print the customer's personal information before printing the discounts.

To accomplish all of the above, the programmer has to create a new hypermodule named ResidentDiscount which is presented in Figure 5.13. As can be seen, there are two concerns created, Contact_Info and Discount. As expected, discount_on_money() and

discount_on_nr_purchases() are merged together using Equate relationship, Braket relationship and mergeByName. First, Braket relationship will make the printAddress() method to be called before discount_on_money() method which will be merged with discount_on_nr_purchases() method.

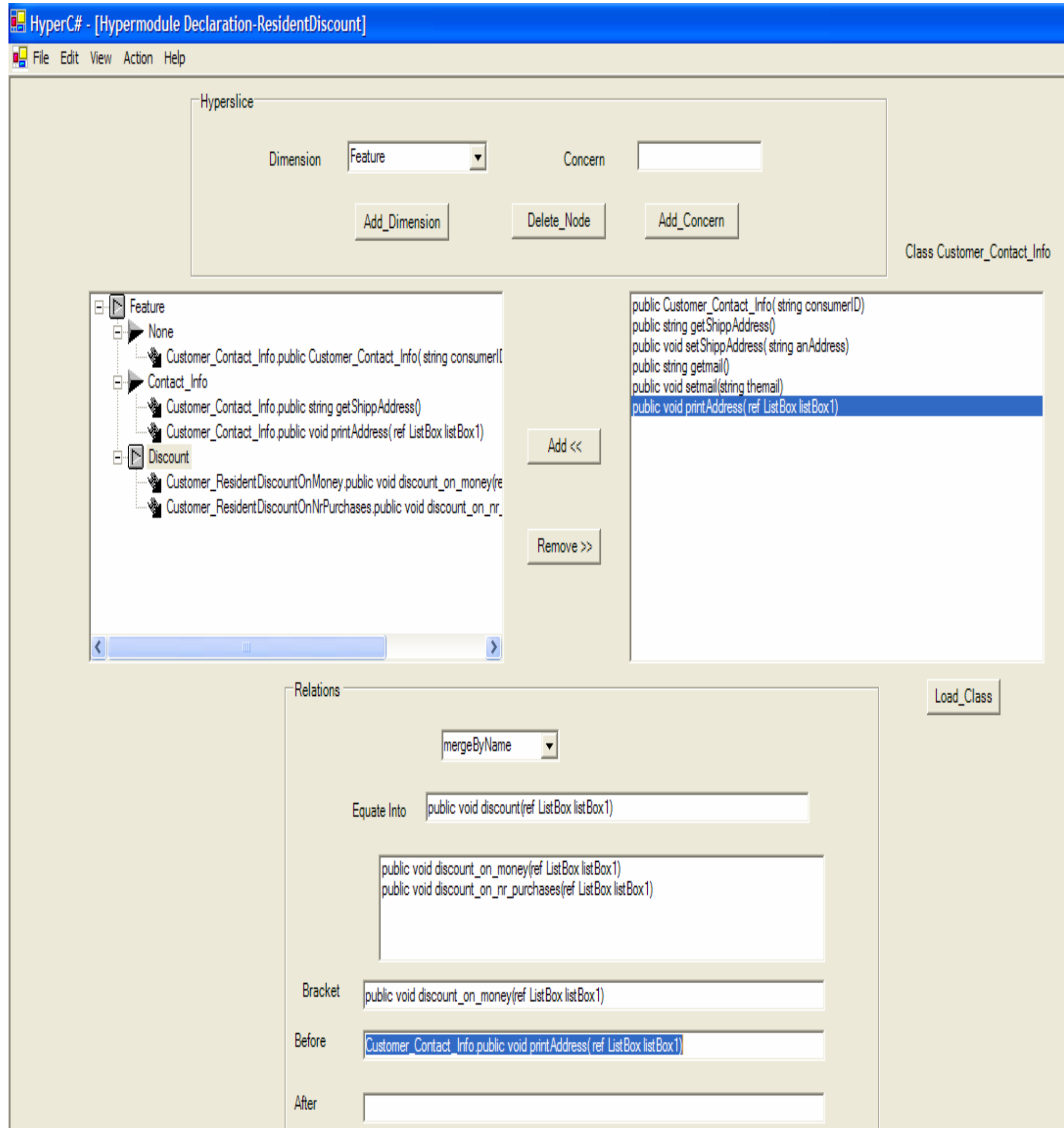
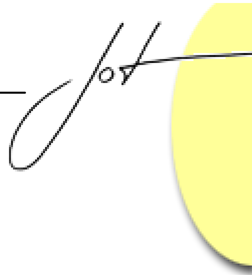



Figure 5.13 ResidentDiscount hypermodule

Figure 5.14 shows the screenshot of the Resident Customer evaluator. It displays the discount of the resident customer Angela Hantelmann and the address where the store will send the discounts.



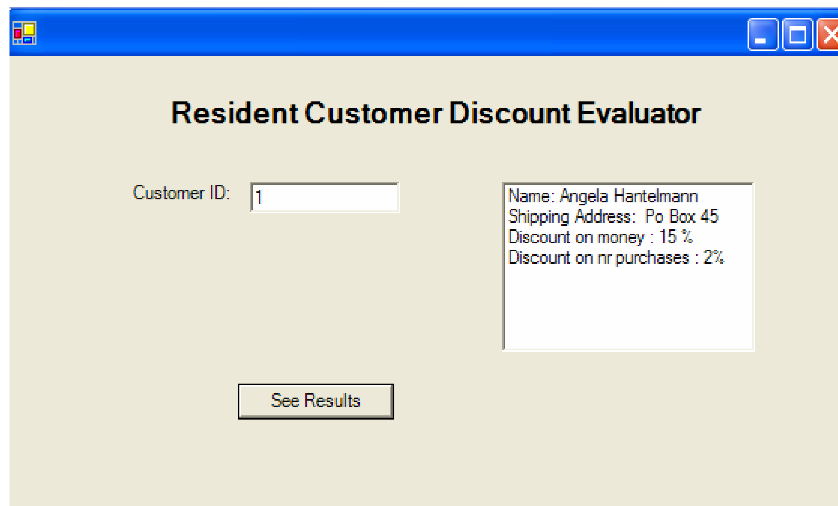



**Figure 5.14 Resident Customer discount evaluator**

## 6 CONCLUSION

HyperC# System enhances C# programming languages by providing a straightforward and manageable approach to software decomposition and integration. Decomposition and integration capabilities offered by HyperC# System contribute to a simplified development and evolution of C# software. Identifying and encapsulating of concerns make software evolution simpler and less costly. Reuse is also facilitated. Integration is simplified because developers need only concentrate on the important interactions among components. All of these (decomposition, integration, reusability) lead to an increasing efficiency in software development which means increased productivity and though, lower cost for software maintenance during its lifecycle.

The important capabilities of HyperC# System are computer aided decomposition and integration. HyperC# offers multiples GUI's that programmers can use to define classes and hypermodules. With the decomposition relationship, programmers are able to identify and modularize the crosscutting concerns as concern classes. These concern classes are used further in the integration process which facilitates mix-and-match of these various concerns to obtain new software. The system provides a weaver that automatically integrates multiple concerns at the same time. The system also provides compilation function and an environment to run the users' programs.

This paper also shows the usefulness of HyperC# System. HyperC# System was used to evolve a Customer evaluator into three new evaluator tools: Resident evaluator tool, Business evaluator tool and Resident Discount evaluator tool. The process of obtaining these new tools using HyperC# is simple. Programmers do not need to write more code. Using GUIs, they just need to decompose the Customer class in more concern classes, create new concern classes and then create the corresponding hypermodules.

There is much work to be done to improve HyperC# System. At this stage, the system works only on classes, so an improvement will be to work on different software components. The current version of the system works only on methods mapping, in the future it will be desirable to extend this mapping to package mapping, class mapping, interface mapping, and fields mapping. Regarding Bracket relationship, it can be improved to support pattern matching, so a set of methods can be specified to be bracket. Another task that can be done in future is to improve the system to be able to work on multi-level hypermodule declarations and integrations.

## REFERENCES

1. Bergmans, L. and Aksit, M., *Composing Crosscutting Concerns Using Composition Filters*. Communications of the ACM, Vol.44, No.10, October 2001, pp.51-57.

2. Channiti, W., *Adding Aspect-Oriented Programming features to Microsoft C#*, MS Project, California State University, Sacramento, Fall 2002

3. Haryono, "*Adding Aspect-Oriented Programming features to Visual Basic.NET by using Multidimensional Separation of Concerns(MDSOC) approach*", MS Project, California State University, Sacramento, Fall 2003

4. Lieberherr, K., Orleans, D., and Ovlinger, J., *Aspect-Oriented Programming with Adaptive Methods*, Communications of the ACM, Vol.44, No. 10, October 2001, pp. 39-41.

5. Netinant, P., Elrad, T., and Fayad, M.E., *A Layered Approach to Build Open Aspect-Oriented Systems*. Communications of the ACM, Vol. 44, No. 10, October 2001, pp. 83-85.

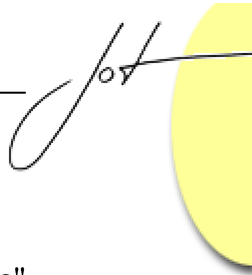
6. Ossher, H., Tarr, P., *Using Multidimensional Separation of Concerns to (Re) Shape evolving Software* ,Communication of the ACM, October 2001/Vol. 44, No. 10, pp 43-50.

7. Ossher, H., Tarr, P., Hyper/J: *Multidimensional Separation of Concerns for Java (Demonstration Proposal)*. IBM Research, 2000

8. Ossher, H., Tarr, P.: *Hyper/J User and Installation Manual*. IBM Research, 2000

9. Sullivan, G.T., *Aspect-Oriented Programming Using Reflection and Metaobject Protocols*. Communications of the ACM, Vol. 44, No. 10, October 2001, pp.95-97.

10. Urban, M., and Spinczyk, O., *Documentation: AspectC++ language Reference*, pure-systems GmbH, June 29, 2004



## About the authors

**Angela Hantelmann** earned her Batchelor Degree in Mathematics from "AL.I.Cuza" University, Romania, in June of 1996. She also obtained a Master Degree in Computer Science from California State University, Sacramento, in May of 2005. She is currently working at Intel, Folsom, California as a software developer.

**Cui Zhang** is a full professor at the Department of Computer Science, California State University Sacramento. Her research and teaching interests include software engineering, object-oriented techniques and methodologies, programming languages, formal methods for software engineering, and formal methods for information assurance and security. She received her Ph.D. in Computer Science from Nanjing University,China, in 1986. She has published more than fifty research papers in international conferences and journals including publishing as primary author in Theoretical Computer Science and Journal of Software Testing Verification and Reliability. She has been also served on the program committees for multiple international conferences.