JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 1, January-February 2010

A Framework for Adding Design by Contract^{TM} to the .NET Object-Oriented Programming Languages

Jennifer Pandolfo, California State University, Sacramento
Cui Zhang, California State University, Sacramento

Abstract

Design contracts can be used in software development to ensure the preservation of assertions for program correctness. This can increase reliability in software design. Design by ContractTM (DBC) was developed by Bertrand Meyer and is supported by the Eiffel programming language. Eiffel provides support for checking preconditions, postconditions, and class invariants automatically at runtime.

Even though DBC has been supported by Eiffel since 1985, other programming languages that offer built-in DBC support are still rare. Redundant efforts have taken place to implement the support of DBC for different object-oriented programming languages. This paper presents the design and implementation of a framework for extending object-oriented programming languages to support DBC. The framework can eliminate the redundant effort for various languages by simplifying the addition of DBC mechanisms for programming language developers.

1 INTRODUCTION

In software development, there has always been the challenge of creating reliable software products. Numerous methodologies have been developed in an attempt to accomplish reliability; one of these is Design by Contract^{TM} (DBC). DBC was first developed by Bertrand Meyer and is supported by the Eiffel programming language [Eiffel07]. DBC specifies and checks design contracts to verify the preservation of certain conditions for program correctness.

Eiffel uses contracts to improve the reliability of software. It does so by checking the preconditions and postconditions of each routine as well as the class invariants. Eiffel also checks other assertions such as loop invariants [Eiffel07]. Obtaining these built-in checking mechanisms gives Eiffel power in assuring program correctness. Even though DBC has been supported by Eiffel since 1985, other programming languages that offer this level of built-in support are still rare. Various efforts have been taken to extend existing object-oriented languages for providing DBC support. Thus far, a new tool has to be developed for each particular object-oriented language [Henne-Wu04, Sorceforge03].

Developing a contract-checking tool for each separate language is an unnecessary effort because these tools [Henne-Wu04, Sorceforge03] perform a common set of functions that are language independent. These can be grouped together and implemented once for use by each language extension. Due to this reuse capability, the utilization of frameworks has become popular in software development [Wikipedia08]. Grouping these common functions leaves only the job of specifying the framework's language dependent portion, which the language designer will have to provide. The combination of language independent parts with language dependent parts forms the framework to support DBC. This paper presents such a framework that simplifies the addition of DBC mechanisms for object-oriented programming language designers. Instead of implementing an entirely new tool to add DBC support to an existing language, the designer can use the framework to extend the programming language. The designer only needs to provide language specific information. Redundant effort is avoided because the language independent features are already implemented.

In today's software development industry, there are numerous object-oriented languages that are widely used, including those in the Microsoft .NET development environment. This environment contains a framework used by each of the .NET languages, which share a Common Language Runtime. Programmers can develop custom applications by combining the unchanging functionality of the framework along with their customized code [Wikipedia08]. Since its release, the Microsoft .NET framework has become one of the most frequently used frameworks. Despite their popularity, the .NET languages do not all contain built-in support for DBC.

The presented framework has been developed for the Microsoft .NET environment. As discussed earlier, there are language dependent and language independent parts to add support for DBC to object-oriented languages. The parts that do not depend on specific languages are unchanged. These parts are utilized to provide DBC support for each object-oriented programming language in .NET. The language dependent components are necessary to extend the language and include various information and a compiler. Language designers must provide these and must also specify how preconditions, postconditions, and class invariants can be verified within their language. The framework uses these specifications to check the conditions for program correctness. The framework provides a mechanism to allow various components to operate together; meaning that the output of a component can be fed as input to other areas of the framework. The language designer should only have to supply the required components, which will function as intended when provided to the framework [Wikipedia08]. The framework also includes a document generator that uses comments and contracts from the code to create documentation.

The goal of this framework is to aid in spreading the use of DBC throughout the software community. More software professionals will be exposed to DBC and to the notions of software correctness and assertion checking. Software correctness and the use of proofs can become more common in industry and not solely in academic environments. This heightened correctness of software in industry will improve the overall quality of software products on the market and also of those written for use within a corporation. This framework not only helps the developer community, but also aids language designers in making programming languages more robust in ensuring correctness and reliability. Mostly, this framework solves the redundant efforts to include DBC support for object-oriented languages. Eliminating this redundancy speeds up industry exposure to DBC with less implementation efforts.

Following this introduction, Chapter 2 of this paper discusses the background and related work. Chapter 3 specifies the design and the architecture of this framework. It reveals how reusable parts interact with the language specific details supplied by the language designer. Chapter 4 explains the framework implementation and gives samples of code. Chapter 5 demonstrates the use of this framework by extending two of the .NET languages to support DBC: C# and Visual Basic .NET. Chapter 6 suggests the future work that can be done to further improve this framework.

2 BACKGROUND AND RELATED WORK

Design by Contract^{TM} (DBC) and the Eiffel Programming Language

A design contract expresses an obligation that software must attain in order for it to be accurate and reliable. Software is considered correct if it meets certain specified conditions and handles unusual situations robustly. Software correctness and reliability can increase with the use of contracts to ascertain a program functions as expected. Aside from checking that software functions as intended, there are other circumstances where DBC can be beneficial in software development. DBC clarifies communication among a design team by stating a program's intentions unambiguously. Either before or during the implementation phase, conditions are explicitly stated that must be met prior to, during, and after the execution of code. This type of clarification can reveal errors early in the design process and gives the involved parties a common understanding of what the code must accomplish [Eiffel07].

DBC not only increases the understanding of software, but can also assist with document generation. The DBC conditions that must be verified give a logical perspective of how the program should function and can provide informative documentation. DBC is also beneficial because it gives an organized process for checking correctness, which may catch more inconsistencies because more areas of code are covered than in random testing methods. "Code coverage, in short, is all about how thoroughly your tests exercise your code base. The intent of tests, of course, is to verify that your code does what it's expected to, but also to document what the code is expected to do" [Koskela04]. Checking conditions at certain points of a method's execution is also effective because verification occurs at the different possible program states [Koskela04].

These stated conditions, commonly called assertions, consist of preconditions, which must be true before methods are executed, postconditions, which must be true after a method's execution when their preconditions are met, and class invariants, which must be true when an object is created as well as before and after each method invocation. Examination of these conditions ensures that the software behaves as specified. In Eiffel, the assertions are automatically checked during program execution to ascertain that variables are within a valid set of values. If the values that Eiffel checks are not within a valid range, the assertions have not been met and the contract has been violated. A violation of contract shows inconsistencies in the program that need correction. Eiffel's built-in automatic contract checking also includes inherited contracts [Eiffel07].

Object-Oriented Programming Language Extension

In order to add support for DBC, various tools have been implemented for the popular object-oriented programming languages. One of these tools, named Contract Sharp[Henne-Wu04], implements DBC for the C# programming language. Contract Sharp provides a GUI for programmers to write code and to specify the contracts associated with that code. With the use of a preprocessor, Contract Sharp combines code and dynamic contract checking into one C# file. Once the source code file is generated, it is compiled, which creates an executable program. There is also a tool to support DBC for the Java programming language [Sorceforge03]. It is called jContractor and enforces contracts at runtime. When using jContractor, programmers can write contracts within the class they apply to or they can put contracts in a separate class that is meant solely for contracts. Like Contract Sharp, jContractor also recognizes preconditions, postconditions, class invariants, and includes support for contract inheritance [Sorceforge03]. Both tools perform similar contract checking functions, but are written for their own different languages. The redundancy in their language extensions can be solved with the use of a software framework.

Software Frameworks

A framework is a software artifact that is reusable. It does not require full functionality because it may be built upon to provide new functionality [Wikipedia08]. The popularity of software frameworks evolved with the release of the Microsoft.NET environment and the Java 2 Platform, Enterprise Edition. The methods of these two frameworks are accessed through an application program interface, or an API. Because some areas of these frameworks can be reused, the low level software details are already accounted for. This allows developers to focus their attention on application specific requirements as well as the software architecture. Frameworks are very useful because they eliminate redundancy in development and yet are also quite flexible.

W. Pree, author of Meta Patterns, describes two areas of software frameworks: hot spots and frozen spots [Wikipedia08]. Hot spots are the areas of the framework where programmers add code to utilize the framework for their particular applications. Frozen spots are the areas that offer reusable unchanging functionality. Frozen spots may not be modified because they are the area of the framework that is reused, therefore eliminating redundancy. These two types of spots give software frameworks their power. Reuse is not helpful if code cannot meet the needs of an application. Hot spots allow flexibility so that reusable code can be utilized for specific applications.

The use of a framework to add support for DBC to the .NET object-oriented programming languages allows extensive code reuse, which saves time and effort for programming language designers. There is no need to write a completely different tool for the DBC extension of each language because the common features are already implemented for use. These parts of the code are the frozen spots. Language designers provide the hot spots to customize the common features to function for their particular programming language. Adding DBC support to programming languages helps ensure code correctness by assessing that the code functions as intended. Because the framework provides this functionality, time and effort are saved when including DBC support for a programming language.

3 FRAMEWORK DESIGN

Identification of Hot Spots and Frozen Spots

In the framework to add support for DBC to object-oriented languages, the parts that can be shared among all languages are considered language independent. They offer unchanging functionality and are the frozen spots of the framework. Frozen spots eliminate redundancy in the implementation of adding DBC support for each language since they can be used by all languages. The parts of the framework that cannot be reused are only functional for a specific language. These parts are language dependent and may not be reused. Because these areas must be adjusted to the needs of a specific language, they are the hot spots of the framework. The language designer must provide hot spots to extend a language for DBC support.

Figure 3.1 gives a list of information that must be supplied for a language extension. Data is organized into two groups: language independent information and language dependent information. The language dependent information listed provides DBC functionality once it is combined with the framework's frozen spots. Once the language specific data is provided, the language is then extended to support DBC. When the language has been extended to support DBC, the extended language programmer can use the automatically created GUI to input code and contracts. The language designer supplies the language syntax and the compiler, written by the compiler writer. These components are provided to the framework along with the source code and contracts written in the extended language. These components, code, and contracts are combined with the frozen spots to generate a syntactically correct source code file with contracts written in the extended language.

Language Dependent	Language Independent
Assignment Syntax	Class Attribute List
Class Access Modifiers	Class Access Modifier
Class Syntax	Class Invariant
Conditional Syntax	Class Name
Compiler Path	Class Other Modifier
Conditional Syntax	Method Access Modifer
Language Name	Method Body
Method Access Modifiers	Method Name
Method Syntax	Method Other Modifier
Other Class Modifiers	Method Parameter List
Program Syntax	Method Return Type
Source File Extension	Precondition
Write To File Syntax	Postcondition

Figure 3.1 - Information Required for DBC Language Extension

System Software Architecture Design

Figure 3.2 illustrates the architecture design for the framework to add support for DBC to the .NET object-oriented programming languages. The language designer provides language specific information including syntactic information for class and method definitions, the path to the compiler, language name, and source code file extension. The syntactic information required includes valid class and method modifiers as well as the syntax for the following: a conditional statement, writing output to a file, an assignment statement, method definition, and class definition. The modifiers are combined with the unchanging functionality of the framework to create a GUI for use by the extended language programmer. The generated GUI is for the specific language and includes the DBC extension.

Syntactic information is used by the framework to generate a preprocessor for the given language definition. When the extended language programmer provides code and contracts, they are supplied to this preprocessor. The preprocessor generates contracts by inserting the code provided by the extended language programmer into the areas specified by the language's syntactic definitions. The contracts provided by the extended language programmer are inserted into the appropriate areas as the code file is generated. A conditional statement to verify the class invariant is inserted at the beginning and also at the end of each method body. Two more conditional statements are inserted into each method body, one of which is inserted at the beginning to check the method precondition and the other is inserted at the end of the method body to verify the method postcondition. Once the preprocessor has processed the provided code, a source code file containing both code and checkable contracts is created in the extended language. This source code file is complied to generate an executable, which will automatically run to check for contract violations. The execution results are recorded in a file for the extended language programmer to examine.

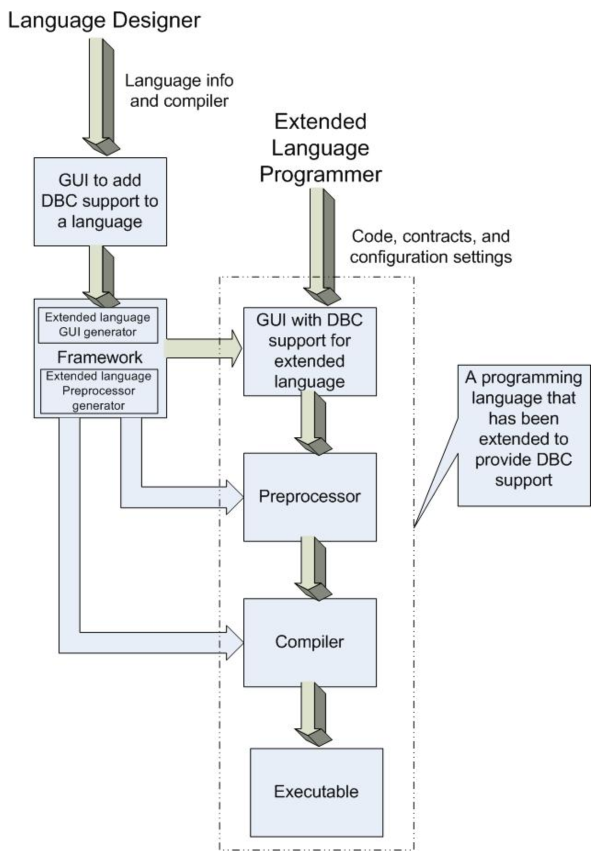

Language Designer
Language info and compiler
GUI to add DBC support to a language
Extended language GUI generator
Framework
Extended language Preprocessor generator
Extended Language Programmer
Code, contracts, and configuration settings
GUI with DBC support for extended language
Preprocessor
Compiler
Executable
A programming language that has been extended to provide DBC support

Figure 3.2 - System Architecture of the Framework

Class-Level Design

Figure 3.3 shows the class level design of the framework to add DBC support to the .NET object-oriented programming languages. The Language class has attributes that contain information about a programming language such as the syntax of a class, method, assignment statement, conditional statement, and also the syntax for writing to a file. Objects of the Language class also contain a path to the compiler, the source code file extension, and the valid modifiers for classes and methods. Accessor and modifier methods are included in the Language class so that objects can be accessed from various forms in the language extension GUI.

Class and Method classes are also provided. These store code and contracts provided by the extended language programmer. Most class and method information is stored in

XML representation for the generation of a final XML document, which occurs once the code and contract specification is complete. Prior to the creation of this document, the XML containing information for each class and method is stored in the XML attribute of each Class and Method object. Class objects are stored in a collection and Method objects are stored in a separate collection. This storage technique simplifies the addition and deletion of both classes and methods from the user interface.

Windows Forms are utilized for the language designer to input language information as well as for the extended language programmer to input class, method, and contract definitions in that particular language. The LanguageDesignerForm class allows language designers to add valid modifiers and access modifiers for classes and methods in the language being extended. An event handler called GenerateToolStripMenuItemClick() uses the provided information to extend the language to include DBC. This event handler writes all language information to a file in XML format. This file is later used to provide language specific information to the language extension GUI. The extended language GUI creates an instance of the LanguageExtensionMainForm. This form contains a language object encapsulated with the language specific information loaded from the XML file. The LanguageExtensionMainForm creates an instance of the LanguageExtensionClassForm when the AddClassButtonClick() method is called. The LanguageExtensionClassForm creates an instance of the LanguageExtensionMethodForm when the method AddMethodButtonClick() is called.

When the CreateSourceCodeFile() method of the Preprocessor class is called, an XML document is generated that contains all code and contracts provided by the extended language programmer. The document is parsed to extract the necessary information and the code with contracts is written into one source code file. The syntactic information for the language specified by the designer is used to ensure the code and contracts are syntactically correct. At this point, a documentation file is generated that includes a description of each class and method as well as their contracts in the program written in the extended language. Once the source code and documentation files have been generated, the extended language programmer can generate an executable file. When the programmer clicks the 'Generate EXE file' button, the GenerateEXEButton_Click() event handler is called. This method compiles the source code file and generates an executable, which is automatically run. The results of the contract checking are output to a text file.

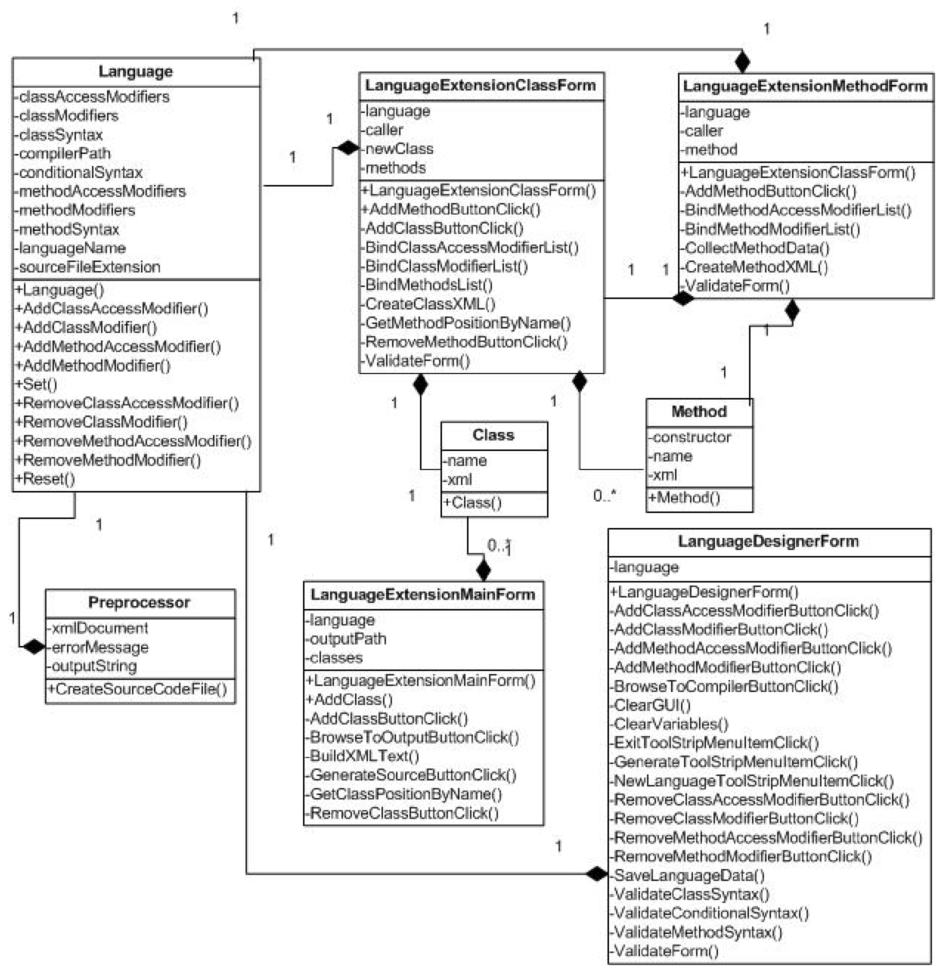

Language
-classAccessModifiers
-classModifiers
-classSyntax
-compilerPath
-conditionalSyntax
-methodAccessModifiers
-methodModifiers
-methodSyntax
-languageName
-sourceFileExtension
+Language()
+AddClassAccessModifier()
+AddClassModifier()
+AddMethodAccessModifier()
+AddMethodModifier()
+Set()
+RemoveClassAccessModifier()
+RemoveClassModifier()
+RemoveMethodAccessModifier()
+RemoveMethodModifier()
+Reset()
LanguageExtensionClassForm
-language
-caller
-newClass
-methods
+LanguageExtensionClassForm()
+AddMethodButtonClick()
-AddClassButtonClick()
-BindClassAccessModifierList()
-BindClassModifierList()
-BindMethodsList()
-CreateClassXML()
-GetMethodPositionByName()
-RemoveMethodButtonClick()
-ValidateForm()
LanguageExtensionMethodForm
-language
-caller
-method
+LanguageExtensionClassForm()
-AddMethodButtonClick()
-BindMethodAccessModifierList()
-BindMethodModifierList()
-CollectMethodData()
-CreateMethodXML()
-ValidateForm()
Class
-name
-xml
+Class()
Method
-constructor
-name
-xml
+Method()
Preprocessor
-xmlDocument
-errorMessage
-outputString
+CreateSourceCodeFile()
LanguageExtensionMainForm
-language
-outputPath
-classes
+LanguageExtensionMainForm()
+AddClass()
-AddClassButtonClick()
-BrowseToOutputButtonClick()
-BuildXMLText()
-GenerateSourceButtonClick()
-GetClassPositionByName()
-RemoveClassButtonClick()
LanguageDesignerForm
-language
+LanguageDesignerForm()
-AddClassAccessModifierButtonClick()
-AddClassModifierButtonClick()
-AddMethodAccessModifierButtonClick()
-AddMethodModifierButtonClick()
-BrowseToCompilerButtonClick()
-ClearGUI()
-ClearVariables()
-ExitToolStripMenuItemClick()
-GenerateToolStripMenuItemClick()
-NewLanguageToolStripMenuItemClick()
-RemoveClassAccessModifierButtonClick()
-RemoveClassModifierButtonClick()
-RemoveMethodAccessModifierButtonClick()
-RemoveMethodModifierButtonClick()
-SaveLanguageData()
-ValidateClassSyntax()
-ValidateConditionalSyntax()
-ValidateMethodSyntax()
-ValidateForm()
1
0..*
0..*

Figure 3.3 - Class Diagram of the Framework

Graphical User Interface Design

The design of the graphical user interface is divided into two parts. The first part consists of a GUI where language designer inputs information for the language extension. This information includes the language name, compiler location, source code file extension, and syntactic information for conditional statements, assignment statements, writing to a file, classes, and methods. These are provided to the framework, which generates various components behind the scenes and also on the front end. If the language designer has provided the necessary information and chooses 'File' > 'Generate GUI for Language Extension', a different GUI is generated. This newly created GUI, displayed in Figure 3.4, is for programmers to enter code and contracts in the language extended by the designer. The GUI shows the modifiers that were provided during the language extension. When a programmer specifies a class, he or she must select one of these modifiers. Using this interface, programmers provide the class invariant, name of the class, attribute list, and also a description, which is used to generate documentation.

When adding a class to the program, at least two methods are required. One of these methods must be a constructor and one of these methods must not be a constructor.

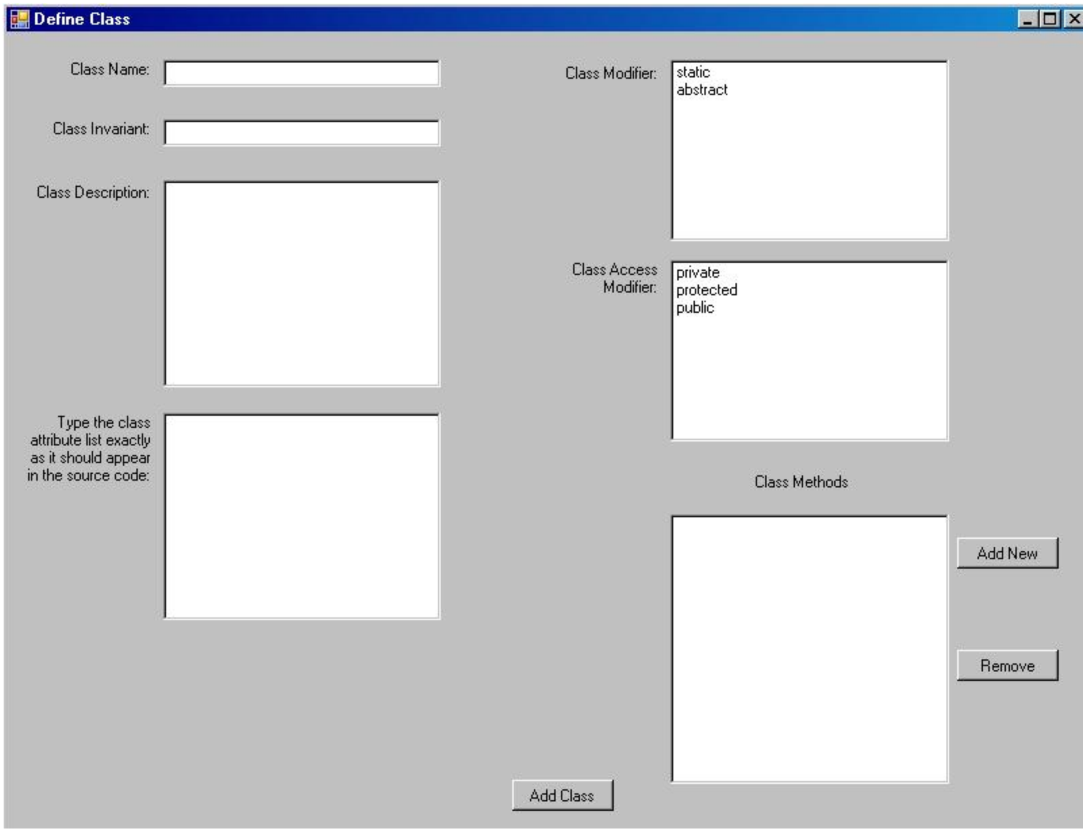


Figure 3.4 - GUI to Define Classes in the Extended Language

When the user clicks the ‘Add New’ button for the class methods, a form to input these methods is displayed and is shown in Figure 3.5.

On this form, the extended language programmer checks a CheckBox indicating if the method is a constructor. The method name, return type, body, and parameter list must also be entered. Programmers choose method modifiers from a ListBox populated with those which are valid in the extended language. Contracts are also input using this form, including the method precondition and postcondition. A method description is included to generate documentation. Once a method is added to the class, its name will appear in a list on the form for inputting classes. When the user enters the required class information and at least two methods, the class may be added to the program. Once added, the class name will appear on the main form.

The main form is displayed in Figure 3.6, which shows the classes that have been defined and also allows the programmer to add namespace and library information to the program. When the programmer clicks ‘Generate Source Code’, an XML file is created by combining the XML of all classes and methods in the program. This file is parsed by the preprocessor to create a file with code and contracts in the extended language. At this point, all class and method descriptions are collected and appended to a string, which is then written to a documentation file. Once the source code and the documentation file have been generated and the 'Generate EXE file' button is clicked, the source code is compiled to generate an executable. This executable is run automatically and the contract checking results are written to a text file.

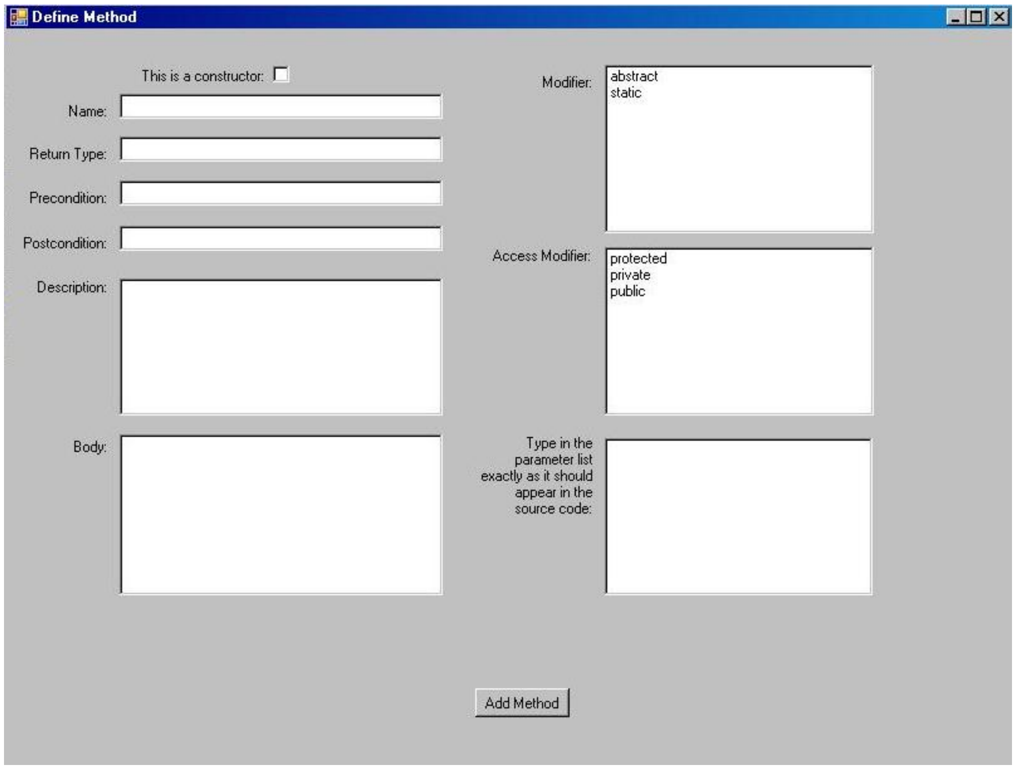
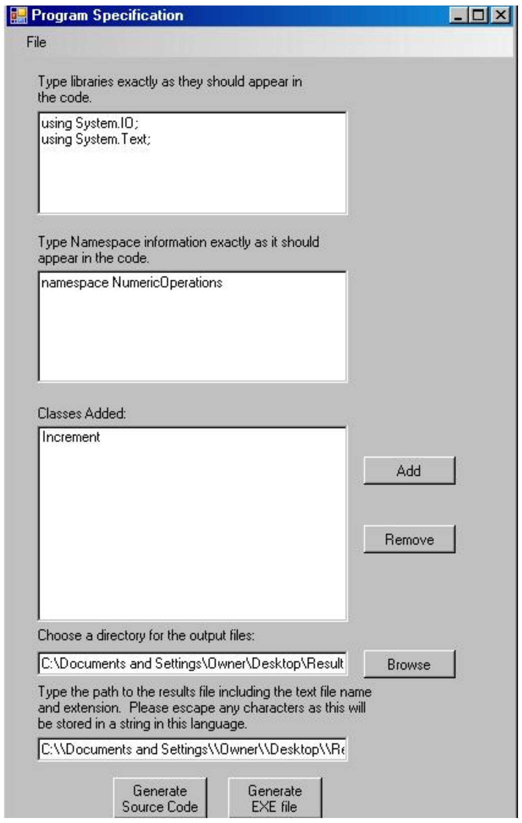


Figure 3.5 - GUI to Define Methods in the Extended Language



Type libraries exactly as they should appear in the code.

```
using System.IO;
using System.Text;
```

Type Namespace information exactly as it should appear in the code.

```
namespace NumericOperations
```

Classes Added:

Increment

Choose a directory for the output files:

C:\Documents and Settings\Owner\Desktop\Result

Type the path to the results file including the text file name and extension. Please escape any characters as this will be stored in a string in this language.

C:\\Documents and Settings\\Owner\\Desktop\\Re

Figure 3.6 - GUI to Add Classes and Generate Output Files

4 FRAMEWORK IMPLEMENTATION

Language Extension

The framework for adding DBC to object-oriented programming languages is implemented in the C# programming language in the .NET environment. The framework contains a GUI, which allows language designers to input syntactic information for the language to be extended. This GUI is a Windows Form called LanguageDesignerForm. A MenuStrip control is used for the top menu and a TabPage control is used for each tab. When the designer clicks the ‘Add’ button to add a modifier, the modifier in the TextBox is added to a list of strings, which are subsequently displayed in a ListBox. AccessModifiers are added in the same manner and are also stored in a list of strings. These lists are attributes of the Language class. The lists are named classAccessModifiers, classModifiers, methodAccessModifiers, and methodModifiers. If the language designer chooses an item in a ListBox of modifiers and clicks ‘Remove’, the modifier will be removed from both the ListBox and from its corresponding List construct.

Once the language designer has provided all required data and selects ‘File’ > ‘Generate GUI for extended language’, the language specific information is validated. If all information is complete and in the correct format, the data is extracted from the interface and is stored in a Language object. The Language object’s attributes are written to a file in XML format, as shown in Figure 4.1. The extended language programmer must run the extended language GUI and specify the path to the XML file containing language specific information. At this time, a new form, named LanguageExtensionMainForm, is instantiated. This form is the GUI that extended language programmers use to provide code and contracts. The Language object, which contains data for the language extension, is passed as a parameter to the LanguageExtensionMainForm’s constructor.

Extended Language

If the extended language programmer wants to add a class to the program, he or she may click the ‘Add Class’ button on the LanguageExtensionMainForm. When this occurs, a new form called LanguageExtensionClassForm is created. Multiple classes can be added using this form. The LanguageExtensionClassForm’s constructor takes the Language object and the LanguageExtensionMainForm as parameters. The LanguageExtensionClassForm is a Windows Form that includes ListBoxes populated with the language specific modifiers provided by the language designer. The modifier lists of the Language object are iterated through and each item is added to its appropriate ListBox. Programmers must select an item from each of these ListBoxes when specifying classes. When the extended language programmer wishes to add a new method, he or she must click the ‘Add New’ button. Once clicked, a new form is instantiated for the collection of method information.

```
private void SaveLanguageData()
{
    StringBuilder sb = new StringBuilder();
    sb.Append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
    sb.Append("<LANGUAGE _Name=\""
    ReplaceSpecialChars(LanguageNameTextbox.Text) + "\" ");

    sb.Append("_CompilerPath=\"" +
    ReplaceSpecialChars(CompilerPathTextbox.Text.Trim()) + "\" ");
    sb.Append("_ConditionalSyntax=\"" +
    ReplaceSpecialChars(ConditionalTextbox.Text) + "\" ");
    sb.Append("_MethodSyntax=\"" +
    ReplaceSpecialChars(MethodSyntaxTextbox.Text) + "\" ");
    sb.Append("_SourceFileExtension=\"" +
    ReplaceSpecialChars(SourceExtensionTextbox.Text.Trim()) + "\" ");
    sb.Append("_WriteToFileSyntax=\"" +
    ReplaceSpecialChars(WriteToFileTextbox.Text) + "\" ");
    sb.Append("_AssignmentSyntax=\"" +
    ReplaceSpecialChars(AssignmentTextbox.Text) + "\" ");
```

```
        sb.Append("_ProgramSyntax=\"" +
        ReplaceSpecialChars(ProgramSyntaxTextbox.Text) + "\" ");
        sb.Append("_ClassSyntax=\"" +
        ReplaceSpecialChars(ClassSyntaxTextbox.Text) + "\">");
        foreach (string accessModifier in language.ClassAccessModifiers){
            sb.Append("<ClassAccessModifier _Value=\"" +
            ReplaceSpecialChars(accessModifier) + "\"/>");
        }
        foreach (string modifier in language.ClassModifiers){
            sb.Append("<ClassModifier _Value=\"" + ReplaceSpecialChars(modifier) +
            "\"/>");
        }

        foreach (string accessModifier in language.MethodAccessModifiers){
            sb.Append("<MethodAccessModifier _Value=\"" +
            ReplaceSpecialChars(accessModifier) + "\"/>");
        }

        foreach (string modifier in language.MethodModifiers){
            sb.Append("<MethodModifier _Value=\"" + ReplaceSpecialChars(modifier)
            + "\"/>");
        }
        sb.Append("</LANGUAGE>");
        string xmlFileName = ExtendedXMLTextbox.Text.Trim() + "\\" +
        LanguageNameTextbox.Text.Trim() + ".xml";
        FileInfo fi = new FileInfo(xmlFileName);
        if (!fi.Exists)
            fi.Create().Dispose();

        StreamWriter txtWriter = File.AppendText(fi.FullName);
        txtWriter.Write(sb.ToString());
        txtWriter.Close();

        MessageBox.Show(xmlFileName + " has been generated");
        this.Close();
    }
```

Figure 4.1 - Code to Store Language Specific Data

This form is called LanguageExtensionMethodForm and allows method specification for that particular class in the extended language. Each time the extended language programmer chooses to add a method, a new instance of this form is created. The LanguageExtensionMethodForm's constructor is supplied the Language object as well as an instance the LanguageExtensionClassForm. Similar to the LanguageExtensionClassForm, the LanguageExtensionMethodForm displays ListBoxes populated with language specific information, however these are filled with valid modifiers for methods instead of those for classes. When the extended language programmer specifies method information and clicks ‘Add Method’, the form is validated to ensure the necessary data is provided. After validation, a method named ReplaceSpecialChars is invoked; at which point all TextBoxes on the form are checked for special characters which include &, >, <, and “. If any of these characters are found, they are replaced in the TextBox with the equivalent XML representation. The XML representation of these characters are as follows: &amp;, &gt;, &lt;, and &quot;. If the special characters are not replaced with their XML equivalent, errors will occur when the generated XML document is loaded for data extraction. The method CollectMethodData is then called; followed by a call to CreateMethodXML. CreateMethodXML inserts the provided data into XML tags, as shown in Figure 4.2. A StringBuilder object is used to append text more rapidly than if a string were used to build the text. The value of the Method object’s XML attribute is assigned to this generated text.

```
private void CreateMethodXML(
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\t\t<METHOD _AccessModifier=\"" +
    MethodAccessModifierListBox.SelectedItem.ToString() + "\" ");
    sb.Append("_OtherModifier=\"" +
    MethodModifierListBox.SelectedItem.ToString() + "\" ");
    sb.Append("_Body=\"" + MethodBodyTextbox.Text.Trim() + "\" ");
    sb.Append("_Description=\"" + MethodDescriptionTextbox.Text.Trim() + "\"
    ");
    sb.Append("_Name=\"" + MethodNameTextbox.Text.Trim()+"\"");
    sb.Append("_Precondition=\"" + PreconditionTextbox.Text.Trim() + "\" ");
    sb.Append("_Postcondition=\"" + PostconditionTextbox.Text.Trim() + "\" ");
    sb.Append("_ReturnType=\"" + ReturnTypeTextbox.Text.Trim() + "\" ");
    sb.Append("_ParameterList=\"" + ParameterListTextbox.Text.Trim() + "\" ");
    sb.Append("_Constructor=\"" + ConstructorCheckbox.Checked + "\"/>\n");
    method.Xml = sb.ToString();
}
```

Figure 4.2 - Code to Store Methods in XML Format

The Method object is added to the List of methods in the LanguageExtensionClassForm and subsequently appears in the ListBox containing methods for that class. In each class, the extended language programmer must provide at least one method that is a constructor and at least one method that is not a constructor. The Method object’s constructor attribute is assigned to the Boolean value of the CheckBox labeled ‘This is a constructor’. After the method has been added to the List, the LanguageExtensionMethodForm is closed.

When a class has at least two methods, the extended language programmer can click the ‘Add Class’ button to add this class to the program. When this button is clicked, the AddClassButton_Click method is called. After validating the form to ensure that all required information is specified, a method named ReplaceSpecialChars searches the text in the TextBoxes for special characters. This method functions in the same manner as the ReplaceSpecialChars method of the LanguageExtensionMethodForm.

```
private void CreateClassXML(ref ClassContainer newClass)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\t<CLASS _Name=\"" + ClassNameTextbox.Text.Trim() + "\" ");
    sb.Append("_AccessModifier=\"" +
    ClassAccessModifierListBox.SelectedItem.ToString() + "\" ");
    sb.Append("_OtherModifier=\"" +
    ClassModifierListBox.SelectedItem.ToString() + "\" ");
    sb.Append("_Invariant=\"" + InvariantTextbox.Text.Trim() + "\" ");
    sb.Append("_AttributeList=\"" + AttributeTextBox.Text.Trim() + "\" ");
    sb.Append("_Description=\"" + ClassDescriptionTextbox.Text.Trim() +
    "\">\n");

    //Add all the methods for this class
    foreach (MethodContainer method in methods)
      sb.Append(method.Xml);

    sb.Append("\t</CLASS>\n");
    newClass.Xml = sb.ToString();
}
```

Figure 4.3 - Code to Store Classes in XML Format

When the special characters are located in the TextBoxes, they are replaced with their valid XML equivalent. Next, the XML representation for the class is generated in the same manner as CreateClassXML method, as shown in Figure 4.3. A StringBuilder object is once again used to rapidly piece together the string of XML. The CreateClassXML method contains a foreach loop in which the CreateMethodXML method is called to append each method's XML to the class body. Once the string of XML is generated for the class, it is stored in the Class object's xml attribute. This Class object can be added to the List of classes in the LanguageExtensionMainForm because the LanguageExtensionClassForm contains an instance of it, giving the form access to this list of Class objects.

Creating Source Code

Once the extended language programmer has finished adding classes to the program, the classes are all stored in the LanguageExtensionMainForm's List of Class objects. When the extended language programmer clicks the 'Generate Source Code' button, at least one Class object has been created. If at least one Class object exists, the list of Class objects is iterated through to build a program string in XML format. This string is built by appending the Xml attribute from each Class object. A StringBuilder object is used to append the text as shown in the figure below. After the string is built, it is then written to an XML file, which will be processed to add contracts when generating the source code file.

```
private string BuildXMLText()
{
    StringBuilder sb = new StringBuilder();
  sb.Append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
  sb.Append("<PROGRAM>\n");
     sb.Append("<LIBRARIES _Text=\"" + LibrariesTextbox.Text +
    "\"/>\n");
    sb.Append("<NAMESPACE _Text=\"" + NamespaceTextbox.Text + "\"/>\n");
    foreach (ClassContainer aClass in classes)
        sb.Append(aClass.Xml);

    sb.Append("</PROGRAM>\n");
    return sb.ToString();
}
```

Figure 4.4 - Code to Build XML String for Extended Language Program

Next, the method CreateSourceCodeFile of the Preprocessor class is called. This method's parameter list includes the Language object, the path of the XML file, the output path for the source code file, and the executable and documentation file's output directory. This output path is specified by the extended language programmer and is retrieved from the GUI. The XML file is then loaded into an XMLDocument object. This load statement is located in a try block for error handling purposes. If the document is syntactically incorrect, a message will be displayed to the extended language programmer. If the document loads successfully, the code selects the LIBRARIES and NAMESPACE nodes. The _Text attribute of these nodes is extracted and appended to the output string used for source code generation. Next, the CLASS nodes are selected and are stored in an XMLNodeList. This list of nodes is iterated through to extract the information necessary to create the program. Each iteration uses the ClassSyntax attribute of the Language class to build the class in the extended language's proper syntax. The class name, modifiers, and attribute list are extracted from the CLASS nodes' attributes _Name, _AccessModifier, _OtherModifier, and _AttributeList. A replace statement is used to place this extracted data into the correct class syntax.

The METHOD nodes are then iterated through. These nodes are children of a CLASS node. Similar to the way the classes are built, methods are formed by extracting information from the XML and using replace statements to put method information in the correct syntax. Contracts are also added to the code at the time methods are built. The method BuildMethodWithContracts is called, which inserts contracts into each method including the precondition, postcondition, invariant check at the beginning of a method, and the invariant check at the end of a method.

Before assembling the contract statements, the *old_* keywords are added to the extended language program. *Old_* keywords are variables added to store the values of a method's variables at the beginning of that method. These are used for comparison with the value of those variables at the end of the method to verify the correctness of the postcondition. For example, a postcondition might ensure that the value of i is one less than its value at the beginning of the method. To perform this check, the value of i is stored in the *old_i* variable at beginning of the method. The postcondition will compare i and old_i to verify that i = *old_i* – 1. To store the old_ variables, the framework parses the postcondition to locate the keyword *old_*. When the position of *old_* is found, the original variable name is extracted from the string at the position after the underscore to the next empty space in the string. The *old_* variable will be assigned the value of the original variable. This framework requires the programmer to initialize the *old_* variables in the method body. The extended language programmer is also responsible for including the comparison in the postcondition of the *old_* variables to the original values. The framework will perform the assignment of the *old_* variables to their original values at the beginning of the method. The programmer must put a comment in the extended language that includes the keyword *PostconditionCheckInit*. This comment must be located after the *old_* variable initialization to represent the location where the framework needs to insert the *old_* variable value assignments.

After any *old_* variables are added to the code, each contract is assembled in a separate method. Figure 4.5 shows the method that builds the precondition. In each of the contract-building methods, a string is initialized to the proper syntax for a conditional statement in the extended language. Next, two strings are initialized with the syntax for writing to a file. These strings store the true block and false block for the conditional statement. The statements to write to a file allow the dynamic contract checking results to be recorded. After these statements are in place, the FileName text is replaced with the path of the file to where the results will be written. The text to be written to a file replaces the TextToWrite placeholder in the true and false block strings. Next, these strings replace the true block and false block placeholders of the conditional statements. The conditional part of the statement is replaced with the precondition for verification. If the precondition is met, the true block will write the appropriate results to the file. If the precondition is not met, the false block will record that the precondition was false in the results file.

```
public static string BuildPrecondition(LanguageContainer language, XmlNode
methodNode, string className, string resultsFile)
{
    string preconditionString = language.ConditionalSyntax;

    string methodName =
    methodNode.Attributes.GetNamedItem("_Name").InnerText.ToStr
    ing().Trim();
```

```
         string methodPrecondition =
        methodNode.Attributes.GetNamedItem("_Precondition").InnerText.ToString().
        Trim();

        string preconditionTrueBlock = language.WriteToFileSyntax;
        string preconditionFalseBlock =language.WriteToFileSyntax;

        preconditionTrueBlock =
        preconditionTrueBlock.Replace(LanguageContainer.FileName,
        resultsFile);

        preconditionTrueBlock =
        preconditionTrueBlock.Replace(LanguageContainer.Text, "Method " +
        methodName + " in Class " + className + " Precondition " +
        methodPrecondition + " evaluated to true");

        preconditionFalseBlock =
        preconditionFalseBlock.Replace(LanguageContainer.FileName,
        resultsFile);

        preconditionFalseBlock =
        preconditionFalseBlock.Replace(LanguageContainer.Text,
        "Method " + methodName + " in Class " + className + " Precondition " +
        methodPrecondition + " evaluated to false");

        preconditionString =
        preconditionString.Replace(LanguageContainer.Condition,
        methodPrecondition);

        preconditionString =
        preconditionString.Replace(LanguageContainer.TrueBlock,
        preconditionTrueBlock);

        preconditionString =
        preconditionString.Replace(LanguageContainer.FalseBlock,
        preconditionFalseBlock);

      return preconditionString;

   }
```

Figure 4.5 - Code to Build Postcondition Verification

Once all four condition checks are built, they are used in piecing together the method source code. The precondition is appended to the start of the method body followed by the beginning invariant check. The method body, extracted from the XML document, is appended to the string after this invariant check. A special case that must be considered is the possibility that methods may contain return statements, which transfer control out of a method. The end invariant check and postcondition must be verified before control is transferred. Because the syntax for a return statement is language dependent, the programmer is responsible for inserting a comment with the keyword *RetValue* on the line before the return statement. The framework will check for the *RetValue* keyword and insert the check for the invariant and postcondition on the next line; before the return statement. If there are no return statements in the method, the postcondition and end invariant check will be appended to the string following the method body.

The method body with contracts is built for each METHOD node. Once the contract and method body strings are combined, the method body contains contracts. The method body is then appended to the class body. At the time the code and contracts are extracted from the XML, the documentation for each class and method is also extracted. These class and method descriptions along with contracts are appended to a string, which is later written to a documentation file. After each method is created for a class, the remaining CLASS nodes are selected from the XMLNodeList of classes and the process of extracting information to build methods and documentation are repeated for those.

```
private void GenerateEXEButton_Click(object sender, System.EventArgs e)
{
    string exeFile = OutputPathTextbox.Text.Trim() + "\\" + SourceFileName +
    ".exe";

    string message = string.Empty;

  System.Diagnostics.Process.Start(language.CompilerPath, "//out:" +
    OutputPathTextbox.Text.Trim() + "\\" + SourceFileName +
    language.SourceFileExtension);

    FileInfo fi = new FileInfo(exeFile);
    if (fi.Exists){
        System.Diagnostics.Process.Start(exeFile);
        message = exeFile + " and " + OutputPathTextbox.Text.Trim() +
        "\\Results.txt generated";
     }
     else
        message = "Executable was not created. Make sure that your code has
        the correct syntax.";

    MessageBox.Show(message);
}
```

Figure 4.6 - Code to Generate and Run Executable

Once the complete program string has been built in the extended language, it is checked for special XML characters. The ReplaceSpecialChars method of the Preprocessor class locates the characters that were inserted into the XML to replace theinvalid XML characters. This method restores them to their original characters. For example, any &gt;

found in the code would be replaced with its original > symbol. After the special characters are replaced, the string is written to a source code file with the proper file extension for the extended language. As mentioned previously, this file is created in a directory specified by the extended language programmer.

After the source code file is created, the programmer can click the ‘Generate EXE file’ button. The code for this event handler is shown in Figure 4.6. When this button is clicked, the compiler provided by the language designer compiles the source code. If the code compiles properly, an executable will be created in the same directory where the source code file is located. If the executable is created successfully, it will run automatically. This executable writes the results of contract checking to a file in the same directory, but its name is specified by the programmer. This results file can be used by the extended language programmer to analyze the contract checking results.

XML Representation

XML representation is valuable to this framework because it is used to store and access data efficiently. When the language specific data is collected from the language designer, it is stored in XML format for use by the extended language GUI. The language information is extracted from this XML file to create the GUI using the syntactic information for that language. As shown in Figure 4.7, the LANGUAGE tag has the following attributes: _CompilerPath, _ConditionalSyntax, _MethodSyntax, _SourceFileExtension, _WriteToFileSyntax, _ProgramSyntax, and _ClassSyntax. These language values are stored as attributes because for each of these items, there can only be one value per language.

```
<?xml version="1.0" encoding="UTF-8"?>
<LANGUAGE _Name="C#"
_CompilerPath="C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc.exe"
_ConditionalSyntax="if(Condition){TrueBlock}else{FalseBlock}"
_MethodSyntax="AccessModifier OtherModifier ReturnType
MethodName(ParameterList)
{MethodBody}"
_SourceFileExtension=".cs"
_WriteToFileSyntax="StreamWriter txtWriter =
File.AppendText(&quot;FileName&quot;);
txtWriter.Write(&quot;TextToWrite\n&quot;);
txtWriter.Close();" _AssignmentSyntax="LeftHandValue = RightHandValue;"
_ProgramSyntax="Namespace{LibrariesClass}"
_ClassSyntax="AccessModifier OtherModifier class ClassName{AttributeList
ClassBody}">
<ClassAccessModifier _Value="public"/>
<ClassAccessModifier _Value=""/>
<ClassModifier _Value="static"/>
<ClassModifier _Value="abstract"/>
<ClassModifier _Value=""/>
<MethodAccessModifier _Value="public"/>
```

```
<MethodAccessModifier _Value="private"/>
<MethodAccessModifier _Value=""/>
<MethodModifier _Value="static"/>
<MethodModifier _Value=""/>
</LANGUAGE>
```

Figure 4.7 - C# Language Specific Information in XML Format

The ClassAccessModifier, ClassModifier, MethodAccessModifier, and MethodModifier nodes are children of the LANGUAGE node because for each language, there can be multiple values for each type of modifier. The language specific data is extracted from this file to create the extended language GUI.

Once the extended language code and contracts have been provided, another XML file is generated. As shown in Figure 4.8, this file contains all code and contracts in XML representation for the extended language. The PROGRAM node contains program information such as libraries and namespaces. Within the NAMESPACE node there can be multiple CLASS nodes. The CLASS nodes have various attributes, which are as follows: _Name, _AccessModifier, _OtherModifier, _Invariant, _AttributeList, and _Description. METHOD nodes are contained within a CLASS node, but are stored as child nodes because a class can contain multiple methods. METHOD nodes have method information containing the following attributes: _AccessModifier, _OtherModifier, _Body, _Description, _Name, _Precondition, _Postcondition, _ReturnType, _ParameterList, and _Constructor. The XML stored for the program is extracted and placed into the appropriate locations according to the language syntax. At this point, a source code file is built with contracts from the data extracted from the XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<PROGRAM>
<LIBRARIES _Text="using System.IO; using System.Text;"/>
<NAMESPACE _Text="namespace NumericOperations"/>
<CLASS _Name="Increment" _AccessModifier="public" _OtherModifier=""
_Invariant="i &gt; 0" _AttributeList="int i;" _Description="Class description">

<METHOD _AccessModifier="" _OtherModifier="" _Body="i =1;"
_Description="Increment constructor description" _Name="Increment"
_Precondition="" _Postcondition="" _ReturnType="" _ParameterList=""
_Constructor="True"/>

<METHOD _AccessModifier="public" _OtherModifier="" _Body="int old_i;
//PostconditionCheckInit
i += 5;
//RetValue" _Description="IncrementByFive description"
_Name="IncrementByFive" _Precondition="i &gt; 0" _Postcondition="i == old_i
+ 5" _ReturnType="int" _ParameterList="" _Constructor="False"/>
```

```
<METHOD _AccessModifier="" _OtherModifier="" _Body="Increment inc = new
Increment();
int s = inc.IncrementByFive();" _Description="Main description" _Name="Main"
_Precondition="" _Postcondition="" _ReturnType="void" _ParameterList=""
_Constructor="False"/>

</CLASS>
</PROGRAM>
```

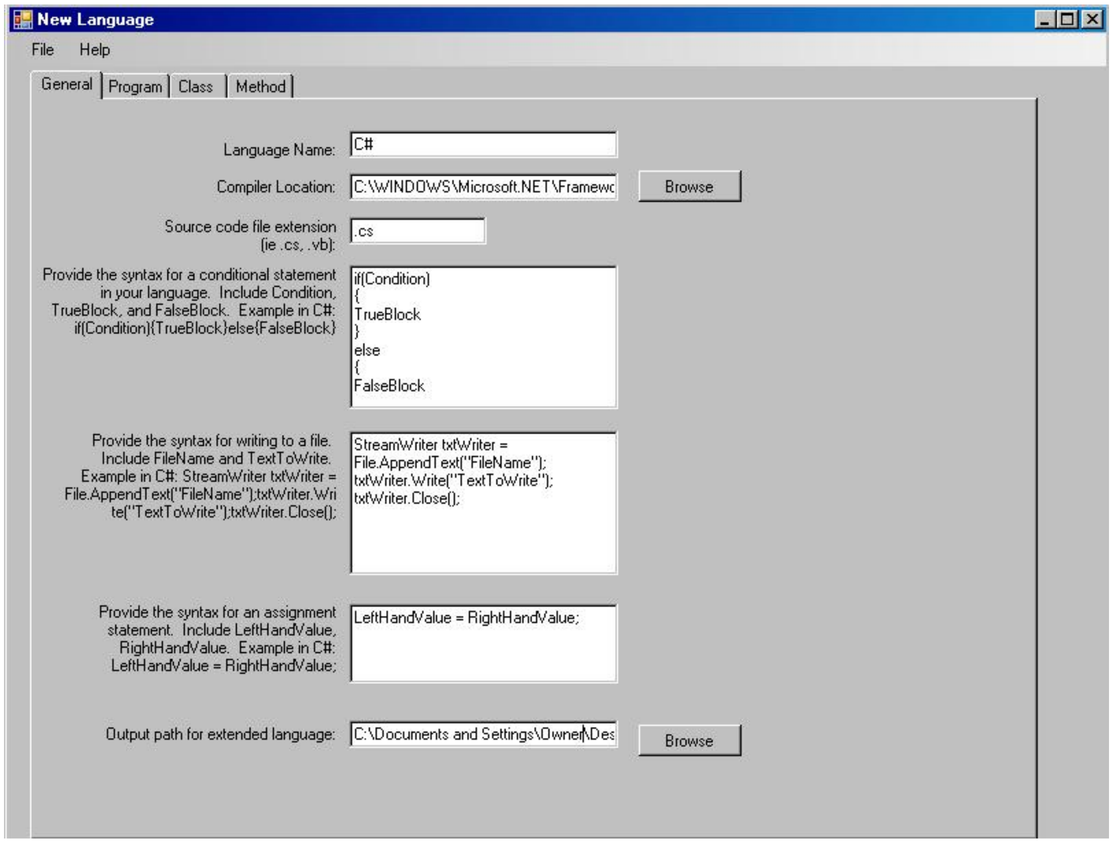
Figure 4.8 - XML Representation of C# Code and Contracts

5 EXTENSION OF C# AND VB.NET TO INCLUDE DBC

This chapter illustrates how to use the framework for the extension of two .NET object-oriented programming languages to include DBC support: C# and VB.NET. The specifications provided for each language extension are shown as well as the use of the GUI generated by the framework to write a program in each language that includes contracts.

Extension of C# to Support DBC

The language designer must supply the language dependent information specific to the C# language. As shown in Figure 5.1, the 'General' tab is used to collect information for the C# programming language. This information includes the language name, path to the compiler, source code file extension, assignment syntax, conditional statement syntax, and the syntax to write to a file. Next, the syntax for a C# program is provided on the 'Program' tab. The program syntax must include the proper locations in a source code file for libraries and namespaces. In C#, the syntax for a valid program is: Namespace{ Libraries Class}.



Language Name: C#

Compiler Location: C:\WINDOWS\Microsoft.NET\Framewc

Source code file extension (ie .cs, .vb): .cs

Provide the syntax for a conditional statement in your language. Include Condition, TrueBlock, and FalseBlock. Example in C#: if(Condition){TrueBlock}else{FalseBlock}

```
if(Condition)
{
TrueBlock
}
else
{
FalseBlock
```

Provide the syntax for writing to a file. Include FileName and TextToWrite. Example in C#: StreamWriter txtWriter = File.AppendText("FileName");txtWriter.Write("TextToWrite");txtWriter.Close();

```
StreamWriter txtWriter =
File.AppendText("FileName");
txtWriter.Write("TextToWrite");
txtWriter.Close();
```

Provide the syntax for an assignment statement. Include LeftHandValue, RightHandValue. Example in C#: LeftHandValue = RightHandValue;

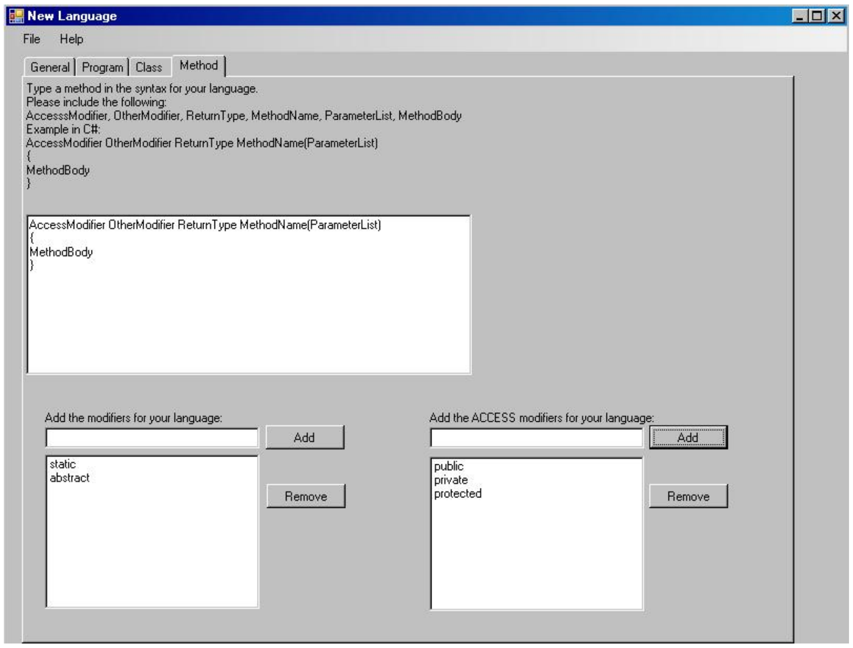
```
LeftHandValue = RightHandValue;
```

Output path for extended language: C:\Documents and Settings\Owner\Des

Figure 5.1 - Language Extension of C#

On the ‘Class’ tab, the C# class syntax specification is input, which is as follows: AccessModifier OtherModifier class ClassName{AttributeList ClassBody}. There are numerous modifiers in C#, but some examples of access modifiers are: private, protected, and public. Other modifiers in C# include abstract and static.

The method syntax for C# is supplied on the ‘Method’ tab, as shown in Figure 5.2. Method access modifiers and other modifiers for C# are also provided using this tab. Once the necessary language specification has been supplied, the language designer can select ‘File’ > ‘Generate GUI for Extended Language’. At this point, the C# language information is written to an XML file. Once the programmer runs the extended language GUI and selects the XML file containing C# information, the language has been extended to include DBC support. C# programmers can add code and contracts using this dynamically generated GUI. Figure 5.3 shows the GUI for program specification in the C# language. The C# programmer can add library and namespace information. A programmer can also add classes and methods to the C# program. If the programmer clicks the ‘Add’ button, the form for class specification will appear, as shown in Figure 5.4.



Type a method in the syntax for your language.
Please include the following:
AccesssModifier, OtherModifier, ReturnType, MethodName, ParameterList, MethodBody
Example in C#:
AccessModifier OtherModifier ReturnType MethodName(ParameterList)
{
MethodBody
}

```
AccessModifier OtherModifier ReturnType MethodName(ParameterList)
{
MethodBody
}
```

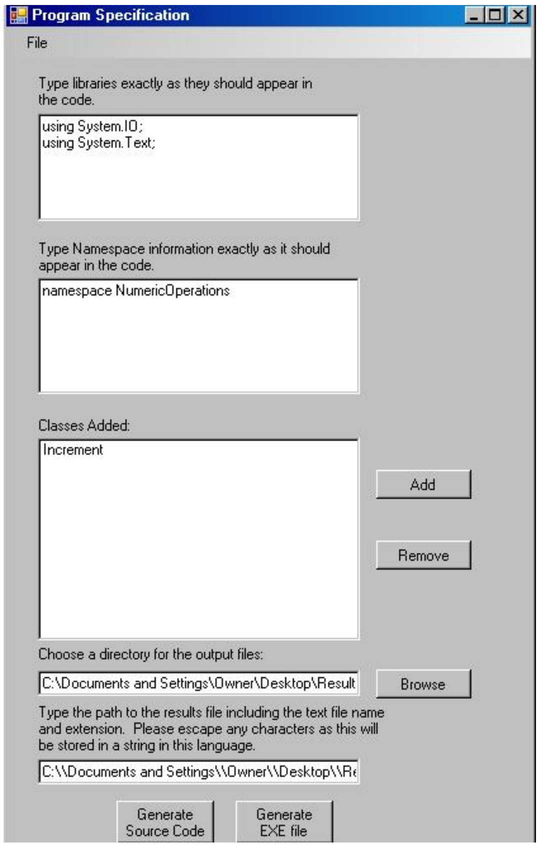
Add the modifiers for your language:

static
abstract

Add the ACCESS modifiers for your language:

public
private
protected

Figure 5.2 - Method Syntax of C#



Type libraries exactly as they should appear in the code.

```
using System.IO;
using System.Text;
```

Type Namespace information exactly as it should appear in the code.

```
namespace NumericOperations
```

Classes Added:

Increment

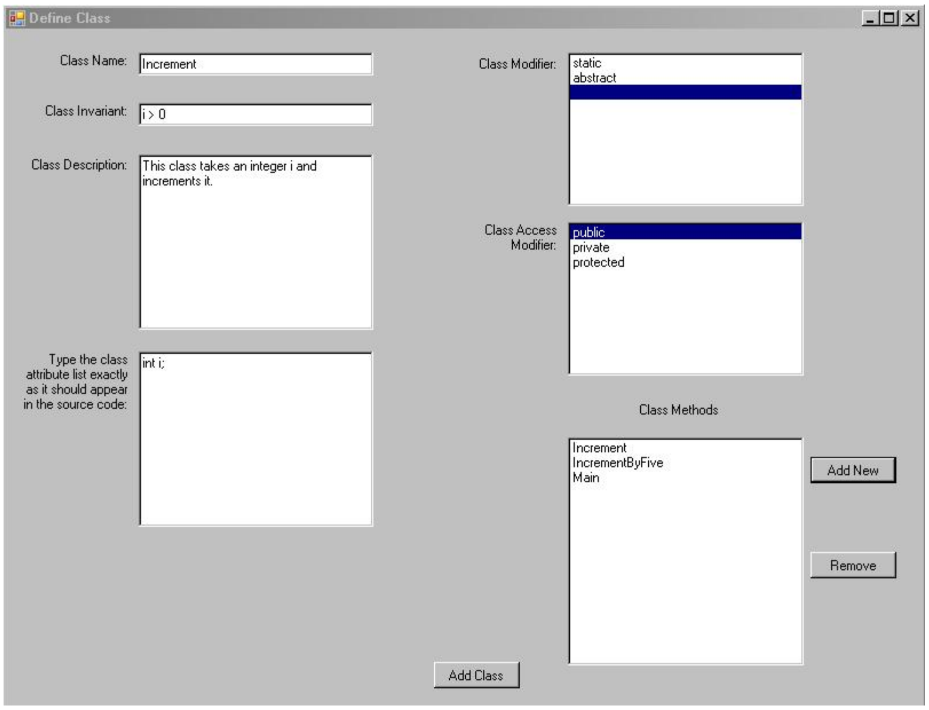
Choose a directory for the output files:

C:\Documents and Settings\Owner\Desktop\Result

Type the path to the results file including the text file name and extension. Please escape any characters as this will be stored in a string in this language.

C:\\Documents and Settings\\Owner\\Desktop\\Re

Figure 5.3 - Program Specification in C#



Class Name: Increment

Class Invariant: i > 0

Class Description: This class takes an integer i and increments it.

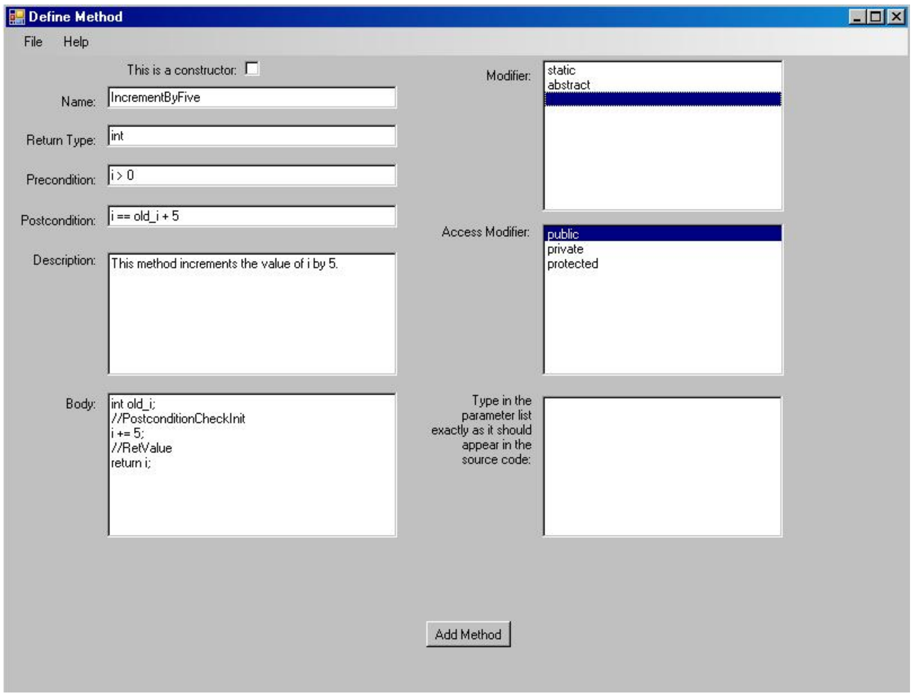
Type the class attribute list exactly as it should appear in the source code: int i;

Class Modifier: static, abstract

Class Access Modifier: public, private, protected

Class Methods: Increment, IncrementByFive, Main

Figure 5.4 - Class Specification in C#



This is a constructor: ☐

Name: IncrementByFive

Return Type: int

Precondition: i > 0

Postcondition: i == old_i + 5

Description: This method increments the value of i by 5.

Body:

```
int old_i;
//PostconditionCheckInit
i += 5;
//RetValue
return i;
```

Type in the parameter list exactly as it should appear in the source code:

Modifier: static, abstract

Access Modifier: public, private, protected

Figure 5.5 - Method Specification in C#

On the form for class specification, the C# programmer supplies the class name, description, attribute list, and chooses modifiers from ListBoxes populated with valid C# modifiers. When the 'Add New' button is clicked, a form to specify C# methods is displayed and is shown in Figure 5.5. The programmer can choose the constructor CheckBox to indicate if the method being specified is a constructor for the class. The programmer also uses this form to give the method name, return type, precondition, postcondition, description, method body, and parameter list. The programmer must also select modifiers from list boxes that are populated with the valid method modifiers in C#.

As shown in Figure 5.5, the C# programmer must initialize the *old_* variables in the method body. The *PostconditionCheckInit* comment is used to indicate where the assignment of the *old_* variables should occur. The programmer is also responsible for comparing the *old_* variable values in the postcondition. Another special comment shown in Figure 5.5 is *RetValue*, which is used to signify a return statement. Since there is a return statement in the method shown in the figure, the postcondition and end class invariant checks are inserted after the *RetValue* comment. Because the framework searches for the *RetValue* comment, the contracts can be verified before control is transferred out of the method. Once all methods and classes have been added to the program, the programmer can select the 'Generate Source Code' button to generate source code with contracts in the C# language.

Figure 5.6 shows the C# source code file generated as a result of using the framework to specify the code and contracts. The proper program syntax was specified by the language designer during the language extension. Library and namespace information provided by the programmer are inserted into their locations in the program. Method and class syntax information provided by the language designer are used to generate the classes and methods in the C# language. The precondition and beginning invariant checks are inserted at the start of the methods. The method bodies are checked for the PreconditionCheckInit and *RetValue* comments to insert *old_* variables and contracts at the appropriate locations in the code. At the time the source code file is generated, a documentation file is also created that contains all contracts, class, and method descriptions. Next, the source code is compiled using the C# compiler and an executable is created in the same directory as the source code. This executable runs automatically to verify contracts. The contract verification results are written to a file for review by the extended language programmer.

Extension of VB.NET to Support DBC

In order to extend VB.NET to include DBC support, the steps that must be taken are similar to the extension of any other object-oriented programming language. Like the extension of C#, the language specific information must be provided for VB.NET. As shown in Figure 5.7, the path to the VB.NET compiler, source code file extension, conditional statement syntax, assignment syntax, and the syntax to write to a file must be provided. After specification of the general language information, the program syntax is supplied for VB.NET, which is as follows: Libraries Namespace Class End Namespace.

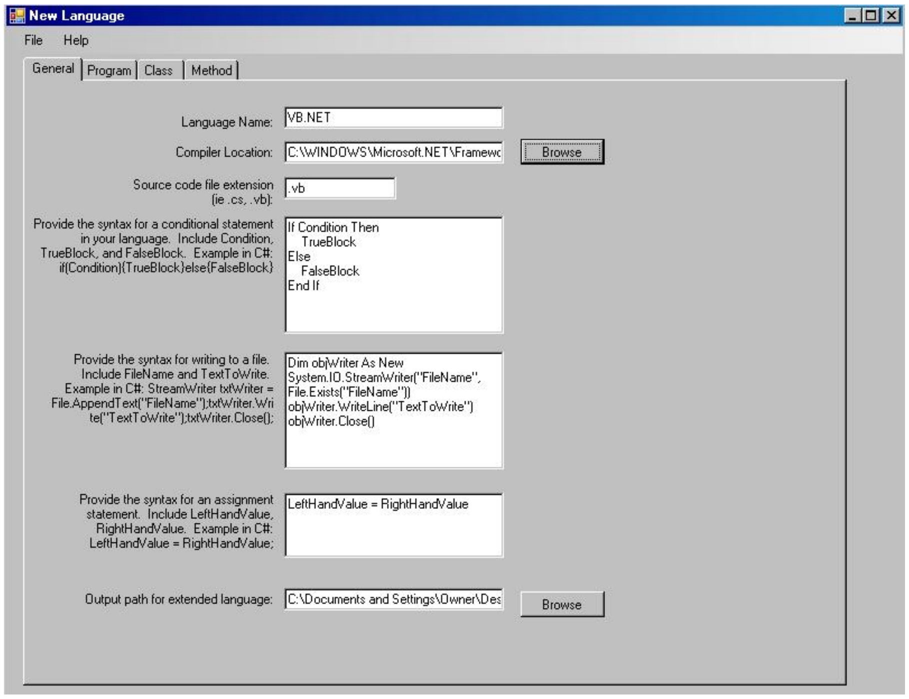
Next, the class syntax for VB.NET is provided as well as some of the valid VB.NET class access modifiers.

```
namespace NumericOperations
{
     using System.IO;
     using System.Text;
     public class Increment
     {
        int i;
        Increment(){
              i = 1;
        }

        public int IncrementByFive(){
           if(i > 0){
              StreamWriter txtWriter =
              File.AppendText("C:\\Documents and
              Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
              txtWriter.Write("Method IncrementByFive in Class Increment
              Precondition i > 0 evaluated to true");
              txtWriter.Close();
           }
           else{
              StreamWriter txtWriter =
              File.AppendText("C:\\Documents and
              Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
              txtWriter.Write("Method IncrementByFive in Class Increment
              Precondition i > 0 evaluated to false");
              txtWriter.Close();
           }

           if(i > 0){
              StreamWriter txtWriter =
              File.AppendText("C:\\Documents and
              Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
              txtWriter.Write("Invariant i > 0 in Class Increment evaluated to
              true at the start of Method IncrementByFive");txtWriter.Close();
           }
           else{
              StreamWriter txtWriter =
              File.AppendText("C:\\Documents and
              Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
              txtWriter.Write("Invariant i > 0 in Class Increment evaluated to
              false at the start of Method IncrementByFive");
              txtWriter.Close();
           }
           int old_i;
           //PostconditionCheckInit
           old_i = i;
```

```
                i += 5;
                //RetValue
                if(i > 0){
                    StreamWriter txtWriter =
                    File.AppendText("C:\\Documents and
                    Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
                    txtWriter.Write("Invariant i > 0 in Class Increment evaluated to
                    true at the end of Method IncrementByFive");
                    txtWriter.Close();
                }
                else{
                    StreamWriter txtWriter =
                    File.AppendText("C:\\Documents and
                    Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
                    txtWriter.Write("Invariant i > 0 in Class Increment evaluated to
                    false at the end of Method IncrementByFive");
                    txtWriter.Close();
                }

                if(i == old_i + 5){
                    StreamWriter txtWriter =
                    File.AppendText("C:\\Documents and
                    Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
                    txtWriter.Write("Method IncrementByFive in Class Increment
                    Postcondition i == old_i + 5 evaluated to true");
                    txtWriter.Close();
                }
                else{
                    StreamWriter txtWriter =
                    File.AppendText("C:\\Documents and
                    Settings\\Owner\\Desktop\\ProjectStuff\\Test3\\Results.txt");
                    txtWriter.Write("Method IncrementByFive in Class Increment
                    Postcondition i == old_i + 5 evaluated to false");
                    txtWriter.Close();
                }
                return i;
            }
            static void Main(){
                Increment inc = new Increment();
                int s = inc.IncrementByFive();
            }
        }
    }
```

Figure 5.6 - Generated C# Source Code and Contracts

Other valid class modifiers are also given. The method syntax is supplied on the 'Method' tab along with some method access modifiers and a few other method modifiers that are valid for VB.NET. After theVB.NET data is given and the language designer selects 'File' > 'Generate GUI for Extended Language', the language information is written to an XML file. Once a VB.NET programmer runs the extended language GUI and selects the VB.NET XML file, the language extension is complete.



Language Name: VB.NET

Compiler Location: C:\WINDOWS\Microsoft.NET\Framewc

Source code file extension (ie .cs, .vb): .vb

Provide the syntax for a conditional statement in your language. Include Condition, TrueBlock, and FalseBlock. Example in C#: if(Condition){TrueBlock}else{FalseBlock}

```
If Condition Then
   TrueBlock
Else
   FalseBlock
End If
```

Provide the syntax for writing to a file. Include FileName and TextToWrite. Example in C#: StreamWriter txtWriter = File.AppendText("FileName");txtWriter.Write("TextToWrite");txtWriter.Close();

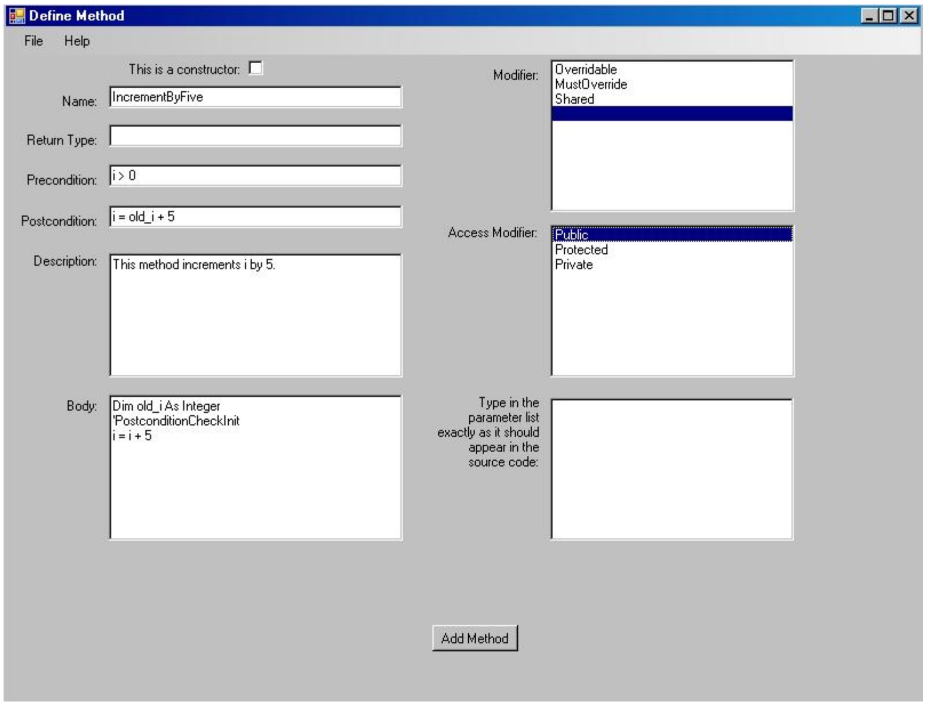
```
Dim objWriter As New
System.IO.StreamWriter("FileName",
File.Exists("FileName"))
objWriter.WriteLine("TextToWrite")
objWriter.Close()
```

Provide the syntax for an assignment statement. Include LeftHandValue, RightHandValue. Example in C#: LeftHandValue = RightHandValue;

```
LeftHandValue = RightHandValue
```

Output path for extended language: C:\Documents and Settings\Owner\Des

Figure 5.7 - Language Extension of VB.NET



This is a constructor: ☐

Name: IncrementByFive

Return Type:

Precondition: i > 0

Postcondition: i = old_i + 5

Description: This method increments i by 5.

Body:

```
Dim old_i As Integer
'PostconditionCheckInit
i = i + 5
```

Modifier: Overridable, MustOverride, Shared

Access Modifier: [✓ Public], Protected, Private

Type in the parameter list exactly as it should appear in the source code:

Figure 5.8 - Method and Contract Definition in VB.NET

At this point, the programmer can supply code and contracts for a program in VB.NET. Library and namespace information is provided and classes are added when the programmer clicks the ‘Add’ button. Once ‘Add’ is clicked, the GUI to add classes to the program is displayed to the programmer, who specifies the class name and invariant, written in VB.NET syntax. A class description and attribute list must also be supplied. The programmer selects class access modifiers and other class modifiers from ListBoxes populated with the class modifiers that are valid in VB.NET.

```
Imports System.IO
Imports System.Text
Namespace NumericOperations
    Public class Increment
      Dim i as Integer

      Public Sub New()
         i = 1
      End Sub

      Public Sub IncrementByFive()
         If i > 0 Then
            Dim objWriter As New System.IO.StreamWriter("C:\Documents
            and Settings\Owner\Desktop\Results.txt",
            File.Exists("C:\Documents and
            Settings\Owner\Desktop\Results.txt"))
            objWriter.WriteLine("Method IncrementByFive in Class
            Increment Precondition i > 0 evaluated to true")
            objWriter.Close()
         Else
            Dim objWriter As New System.IO.StreamWriter("C:\Documents
            and Settings\Owner\Desktop\Results.txt",
            File.Exists("C:\Documents and
            Settings\Owner\Desktop\Results.txt"))
            objWriter.WriteLine("Method IncrementByFive in Class
            Increment Precondition i > 0 evaluated to false")
            objWriter.Close()
         End If
         If i > 0 Then
            Dim objWriter As New System.IO.StreamWriter("C:\Documents
            and Settings\Owner\Desktop\Results.txt",
            File.Exists("C:\Documents and
            Settings\Owner\Desktop\Results.txt"))
            objWriter.WriteLine("Invariant i > 0 in Class Increment
            evaluated to true at the start of Method IncrementByFive")
            objWriter.Close()
         Else
            Dim objWriter As New System.IO.StreamWriter("C:\Documents
            and Settings\Owner\Desktop\Results.txt",
            File.Exists("C:\Documents and
            Settings\Owner\Desktop\Results.txt"))
```

```
                objWriter.WriteLine("Invariant i > 0 in Class Increment
                evaluated to false at the start of Method IncrementByFive")
                objWriter.Close()
            End If
            Dim old_i As Integer 'PostconditionCheckInit
            old_i = i
            i = i + 5
            If i = old_i + 5 Then
                Dim objWriter As New System.IO.StreamWriter("C:\Documents
                and Settings\Owner\Desktop\Results.txt",
                File.Exists("C:\Documents and
                Settings\Owner\Desktop\Results.txt"))
                objWriter.WriteLine("Method IncrementByFive in Class
                Increment Postcondition i = old_i + 5 evaluated to true")
                objWriter.Close()
            Else
                Dim objWriter As New System.IO.StreamWriter("C:\Documents
                and Settings\Owner\Desktop\Results.txt",
                File.Exists("C:\Documents and
                Settings\Owner\Desktop\Results.txt"))
                objWriter.WriteLine("Method IncrementByFive in Class
                Increment Postcondition i = old_i + 5 evaluated to false")
                objWriter.Close()
            End If
            If i > 0 Then
                Dim objWriter As New System.IO.StreamWriter("C:\Documents
                and Settings\Owner\Desktop\Results.txt",
                File.Exists("C:\Documents and
                Settings\Owner\Desktop\Results.txt"))
                objWriter.WriteLine("Invariant i > 0 in Class Increment
                evaluated to true at the end of Method IncrementByFive")
                objWriter.Close()
            Else
                Dim objWriter As New System.IO.StreamWriter("C:\Documents
                and Settings\Owner\Desktop\Results.txt",
                File.Exists("C:\Documents and
                Settings\Owner\Desktop\Results.txt"))
                objWriter.WriteLine("Invariant i > 0 in Class Increment
                evaluated to false at the end of Method IncrementByFive")
                objWriter.Close()
            End If
        End Sub

        Shared Sub Main()
            Dim inc As New Increment()
            inc.IncrementByFive()
        End Sub
    End Class
End Namespace
```

Figure 5.9 - Generated VB.NET Source Code and Contracts

The programmer adds methods by clicking the ‘Add New’ button.

As shown in Figure 5.8, the programmer specifies method information written in VB.NET. The programmer inputs the method name, return type, parameter list, description, and whether the method is a constructor. Method access modifiers and other modifiers are selected from the ListBoxes, which are populated with valid VB.NET method modifiers. Method contracts are provided in VB.NET, which include the precondition and postcondition. The postcondition supplied by the programmer may contain a comparison of the *old_* variables with their new value. As shown Figure 5.8, the body can contain the declaration of the *old_* variables and the *PostconditionCheckInit* comment. As explained previously, this comment signifies the location of the *old_* variable declarations. The framework inserts the assignment of the *old_* variables to their original value after this comment. The values of the *old_* variables are verified in the postcondition at the end of the method. Another comment in the method body with special meaning to the framework is the *RetValue* comment. As explained in the C# language extension, this comment signifies a return statement. The framework inserts the postcondition and end invariant before the return statement for contract verification. Inserting the conditions at this location ensures they are verified before control is transferred out of the method.

After the programmer provides code and contracts for the VB.NET program, the ‘Generate Source Code’ button can be clicked. When this occurs, the preprocessor builds the code file by inserting namespace and library information into VB.NET syntax. Contracts written in VB.NET are inserted at the appropriate positions in each method. The precondition and beginning invariant check are inserted before the method body. The method body is then searched for the *RetValue* comment. If the *RetValue* comment appears in the method body, the preprocessor will insert the end invariant check and postcondition before the return statement. If the *RetValue* comment is not found, the preprocessor will insert the end invariant check and postcondition after the method body. The preprocessor also checks the method body for the *PostconditionCheckInit* comment. If this comment is found, the *old_* variables found in the postcondition are initialized and are then assigned to their original value at the beginning of the method. At the time the source code file is built, the contracts, class, and method descriptions are extracted from the XML. These contracts and descriptions are appended to a string, which is written to a documentation file. Classes and methods with all contracts are then added to the source code file. The source code generated by this example is shown in Figure 5.9. After the code is compiled and the resulting executable runs, the contract verification results are written to a text file.

6 FUTURE WORK

As presented, the framework to add DBC support to object-oriented programming languages is by no means perfect. However, it is a strong beginning to eliminate the redundant effort to add contract checking to each object-oriented language. Language designers can use this framework to extend their language by allowing programmers to build a software artifact in the extended language containing both code and dynamically checkable contracts. A documentation file is also generated at this time, which includes a description of each class, method, class invariant, method precondition, and method postcondition in the program. When the extended language programmer clicks the 'Generate EXE file' button, the source code file containing contracts is compiled, which generates an executable. The framework runs this executable automatically creating a text file that contains contract checking results. Therefore, the effort needed to extend an object-oriented programming language to include DBC has been greatly reduced with the construction of this framework. Using C# and VB.NET, Chapter 5 illustrated the simplification of these language extensions. With the guidance of the help menus, only a few language specific items must be supplied and the language has been extended to include the support for DBC.

In addition to testing the extension of more languages using this framework, various features can be added to increase the framework's power and usability. Even though the framework functions for object-oriented programming languages, the current version does not support contract inheritance. Similar to the inheritance of attributes and methods, contracts from a parent class are also inherited. In order for parent contracts to be verified logically, language designers will need to specify the syntax for inheritance in the language that is being extended.

The researchers extending this framework will need to add code that will logically OR the precondition of the current class with the precondition of the parent class. This logical OR weakens the precondition. The extended language programmer also needs to add code that will logically AND the postcondition of the current class with the postcondition of the parent class. Therefore, inheritance strengthens the postcondition. Inheritance of class invariants functions in the same manner as the inheritance of postconditions. The researchers extending this framework will also need to logically AND the current class invariant with the parent class invariant. The current framework inserts conditional statements to check the specified contracts. These conditional statements will need to be modified to include the contracts for the parent class.

Another function that can be added to the framework is the ability to specify the level of dynamic contract checking. This feature gives the programmer the option to choose the combination of contracts to be checked for the program. The framework currently checks and reports the result of all preconditions, postconditions, and invariants. Allowing the extended language programmer to choose the checking level will increase the flexibility of the contract checking mechanism.

REFERENCES

[Eiffel07] Eiffel Software. Eiffel in a Nutshell. [Online]. 1985-2007. Available: http://archive.eiffel.com/eiffel/nutshell.html

[Henne-Wu04] Henne-Wu, Rachel, William Mitchell, and Cui Zhang: "Support for Design By Contract™ in the C# Programming Language", Journal of Object Technology vol. 4, no. 7, September-October 2004, pp. 65-82.

[Sorceforge03] Sourceforge. What is jContractor? [Online]. 2003 Available: http://jcontractor.sourceforge.net/

[Wikipedia08] Wikipedia. (2008, Aug.). Software Framework. [Online]. Available: http://en.wikipedia.org/wiki/Software_framework#_note-7

[Koskela04] Koskela, L. Introduction to Code Coverage. Accenture Technology Solutions. [Online]. 2004, Jan. Available http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html

[Grassman96] Grassman, W. and J. Tremblay. Logic and Discrete Mathematics. New Jersey: Prentice Hall, Inc., 1996, pp. 482-489.

[Markiewicz01] Markiewicz, M. and C. Lucena. Object Oriented Framework

Development. [Online]. 2001 Available: http://www.acm.org/crossroads/xrds7-4/frameworks.html

About the authors

Jennifer Pandolfo earned her Master's Degree in Computer Science from California State University, Sacramento, in 2008 and her Bachelor's Degree in Computer Science from California State University, Sacramento in 2004. She is currently employed by Old Republic Title Information Concepts, in Roseville,California, as Ecommerce Manager. She can be contacted at pandolfojennifer@gmail.com.

Dr. Cui Zhang is a full professor in the Department of Computer Science, California State University Sacramento. Her research and teaching interests include software engineering, programming language theories and paradigms, and formal methods for software engineering and for information assurance and security. She can be reached at zhangc@ecs.csus.edu.