RICE UNIVERSITY

Dr. C#: A Pedagogic IDE for C# Featuring a Read-Eval-Print-Loop

by

Dennis Lu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Robert Cartwright, Chair
Professor of Computer Science

Joe Warren
Professor of Computer Science

Dan Wallach
Assistant Professor of Computer Science

Dung X. Nguyen
Lecturer, Computer Science

Houston, Texas

April, 2003

Dr. C#: A Pedagogic IDE for C# Featuring a Read-Eval-Print-Loop

Dennis Lu

Abstract

As the primary programming language of the Microsoft .NET platform, C# will play a significant role in software development for the foreseeable future. As the language rapidly gains popularity in industry, tools made for C# development focus on the professional programmer, while leaving the beginning computer science student behind.

To address this problem, we introduce Dr. C#, a simple lightweight development environment with an integrated, interactive Read-Eval-Print-Loop (REPL). Dr. C# helps flatten the learning curve of both the environment and the language, enabling students to quickly learn key elements of the language and focus more easily on concepts. Dr. C# thus serves not only as a learning tool for beginner students but also as a teaching tool for instructors. The editor is based on an open source IDE called SharpDevelop. This thesis describes the implementation of Dr. C# focusing primarily on building the REPL and integrating with SharpDevelop.

Acknowledgments

I would like to thank my advisor Professor Robert “Corky” Cartwright for giving me the opportunity and autonomy to develop Dr. C#. I would also like to thank Professor Dung Nguyen for acting as co-advisor and for his years of guidance and support.

This project would not have been possible without the generous support of Microsoft Research.

Many thanks also to Mathias Ricken for his substantial development and integration efforts and Nick Becker for his testing efforts in helping in the development of Dr. C#.

Contents

Illustrations

Chapter 1

Introduction

The C# programming language is rapidly gaining acceptance in the developer community [25]. Microsoft has "bet the ranch" with its billion plus dollar investments into the .NET platform and has made it clear that C# is the premier language for targeting .NET [16]. Companies like HP are investing millions to train consultants and push the .NET platform [20], and businesses around the world are starting to listen and buy into what the .NET platform can offer them [8].

Currently, though, the Java platform has a strong foothold in both industry and academia, but C# offers many of the same and even improves on some of the features of Java [14]. Even Java proponents realize that C# has the potential to replace Java as the object oriented language of choice among professional developers [27]. Universities are also starting to open up to the idea of using C# to teach their courses [23].

With all this momentum behind it, C# and .NET are here to stay, and the demand for knowledgeable C# developers will only rise. Whether through student demand, business pressure, or simply the compelling features of and research possibilities offered by the C# language, academia will soon find itself needing to teach C#.

No matter how much lecturing a professor does, in the end the student will have to learn the concepts and programming skills by sitting himself in front of a computer and writing some code. Currently, given the set of tools available, if a students wants to do some programming in C#, he will have to find a Microsoft Windows computer with the .NET Framework and Visual Studio .NET installed.

The preparation needed to set up a system like this for C# development can be a hassle. While setting up the .NET Framework is straightforward, obtaining Visual Studio .NET either requires some special program with Microsoft to obtain a free copy or requires a large sum of cash. Either way, once the student opens up Visual Studio .NET for the first time, he will encounter a large and powerful development environment designed to help professional programmers quickly build .NET web services. The student will have to create things like a Solution and decide on a type of Project before he can even write a simple “Hello World” program.

Subsequent uses of Visual Studio .NET will present the user with wizards, dynamic help, property pages, and many more features that make web service development easier. To beginning students however, web service development is hardly on the forefront of their minds. All these features mean nothing to someone just learning the language. In fact, many times, these professional productivity tools can turn out to be counter-productive if focus is removed from learning concepts to learning how to use development environment.

Furthermore, Visual Studio .NET and the various other more minor development tools for C# lock the student in the typical edit-compile-run iterative development cycle. There is no element of interactivity between the student and his code. There is a need for a better solution.

To address these issues, we introduce Dr. C#. Dr. C# is a lightweight integrated development environment (IDE) designed from the very beginning with the student in mind. The most important contribution to the world of C# development is the addition of a C# interpreter integrated into an IDE as a Read-Eval-Print-Loop (REPL). This brings interactivity to the world of C#, a feature that many developers find useful in other languages. Moreover, the student is presented with a very simple and intuitive user interface free of superfluous windows and other so called productivity features that frequently distract from more than help in development. The student is now free to focus his efforts on the language and not get bogged down in learning how to use an editor.

This thesis describes the effort to produce Dr. C#. Chapter 2 details the reasons for producing Dr. C#. Chapter 3 examines previous work related to Dr. C#. Chapter 4 provides an in depth explanation of the development and architecture of the REPL. Chapter 5 talks about the open source editor SharpDevelop as the foundation of the editor. Chapter 6 shows how the two parts are integrated into the final product. Chapter 7 discusses possible future extensions and improvements for each part of Dr. C#. Chapter 8 concludes.

Chapter 2

Rationale and Motivation

The primary motivation for developing Dr. C# is to make a simple, interactive, and free development environment for the beginning computer science student to program in C#. C# is the new programming language rapidly gaining popularity in the software industry and making inroads into academia as well [23]. Software giant Microsoft has put substantial capital in to making C# the premier language for it .NET initiative.

Looking at the field of development environments for C#, the beginning computer science student has several hurdles to jump before he even begins writing his first program. His choices for development are Microsoft Visual Studio .NET and a bevy of open source tools. Each of which, however, do not provide a way for the student to interact with the code, and examining the programs available, one will see the need for something new.

2.1 Visual Studio .NET

Microsoft Visual Studio .NET is the premier IDE for C# and is currently the flagship product for Microsoft in its .NET initiative. According to a Microsoft web site, .NET is a set of software technologies for connecting information, people, systems, and devices though XML Web Services [4]. As one can imagine, the ability to develop these services requires some advanced understanding of programming. To attract developers to .NET, Microsoft has made considerable effort to make the task of developing web services as easy as possible [24] by including a wide variety of features and wizards to perform various tasks as well as making the editor highly configurable to the developer's tastes. As a result Visual Studio .NET is a very powerful IDE. In fact, it is a little too powerful.

These features may make the professional software developer more productive, but to a beginning student just learning what object oriented programming is, features like wizards just get in the way. Wizards try to make mundane and frequently done tasks easier by abstracting out the details and are best used when the user has an understanding of what exactly is being abstracted out. The problem for students is what happens when there is a bug. Without a low level understanding of what his program is doing, a student will be lost.

The high configurability of Visual Studio .NET also does not help the learning process. As can be seen in Figure 2.1, the user is presented with an interface with many different windows and functions. By default, the user gets help, solution, task, output, and property windows. Having all these windows around leaves little room for displaying the actual code. An advanced user will quickly realize that even he does not need all of them and will find a way to dispose of them. A beginner, however, probably does not know what he needs and may just keep the superfluous windows around. The drawback to this is a confusing and intimidating interface presenting information that the student fails to comprehend and detracts attention from the code, which is the most important part.

Other features in Visual Studio .NET can confuse even advanced users. For example, code may be grouped into containers called Projects. However, there is also a container called a Solution. Solutions can contain one or multiple projects. However, the wizard one must go through to create Solutions and Projects always names the two the same. Other kinds of terminology like the ability to both Build and Rebuild Projects and Solutions can lend to confusion, as well.

These features do not come cheap, in more ways than one. The Professional version of Visual Studio .NET retails for $1,079 and has an installation image of over 1GB. It also requires a somewhat fast computer to run decently. This is indeed one huge roadblock for students.

To solve the monetary cost problem of obtaining Visual Studio .NET, Microsoft developed an Academic version of Visual Studio .NET, which is distributed freely to many university students. It roughly targets the same audience as Dr. C#. How-

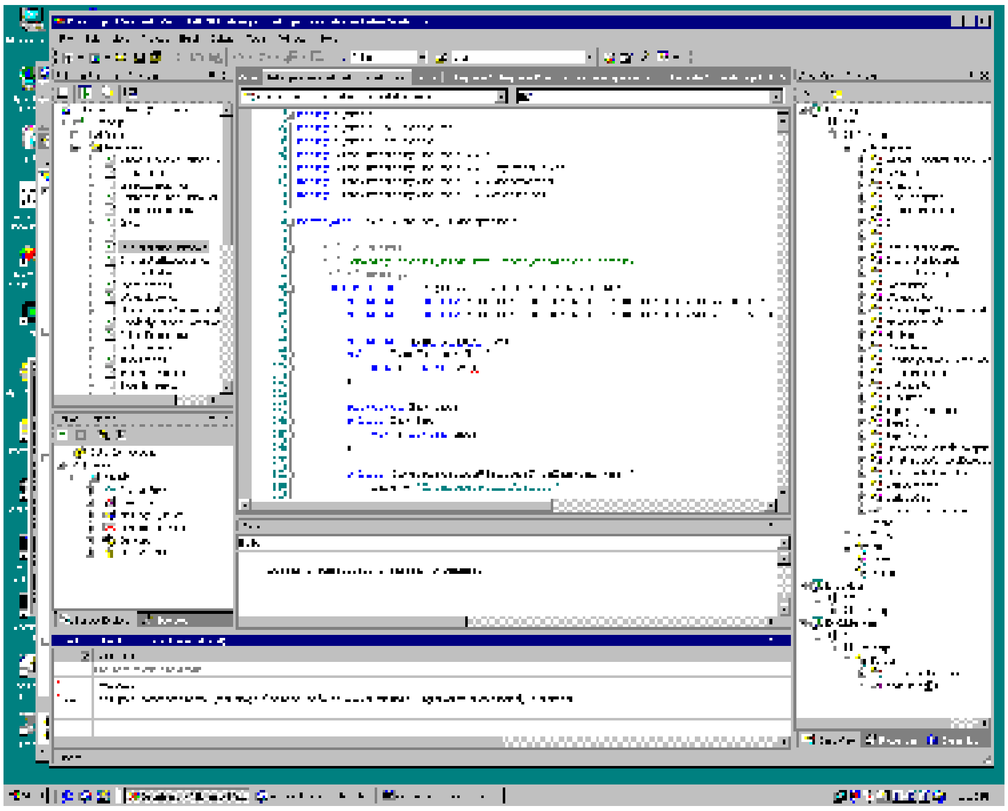


Figure 2.1 : Visual Studio .NET user interface

ever, after installing this version, one quickly realized that it is exactly the same as the Professional version except for a couple extra features that aid instructors in course organization, like the ability to submit code electronically. These additions aid the professors and class administrators more than it helps students. No attention was placed on simplifying the interface or helping students focus on learning programming concepts. In addition, this version is not available to all students everywhere.

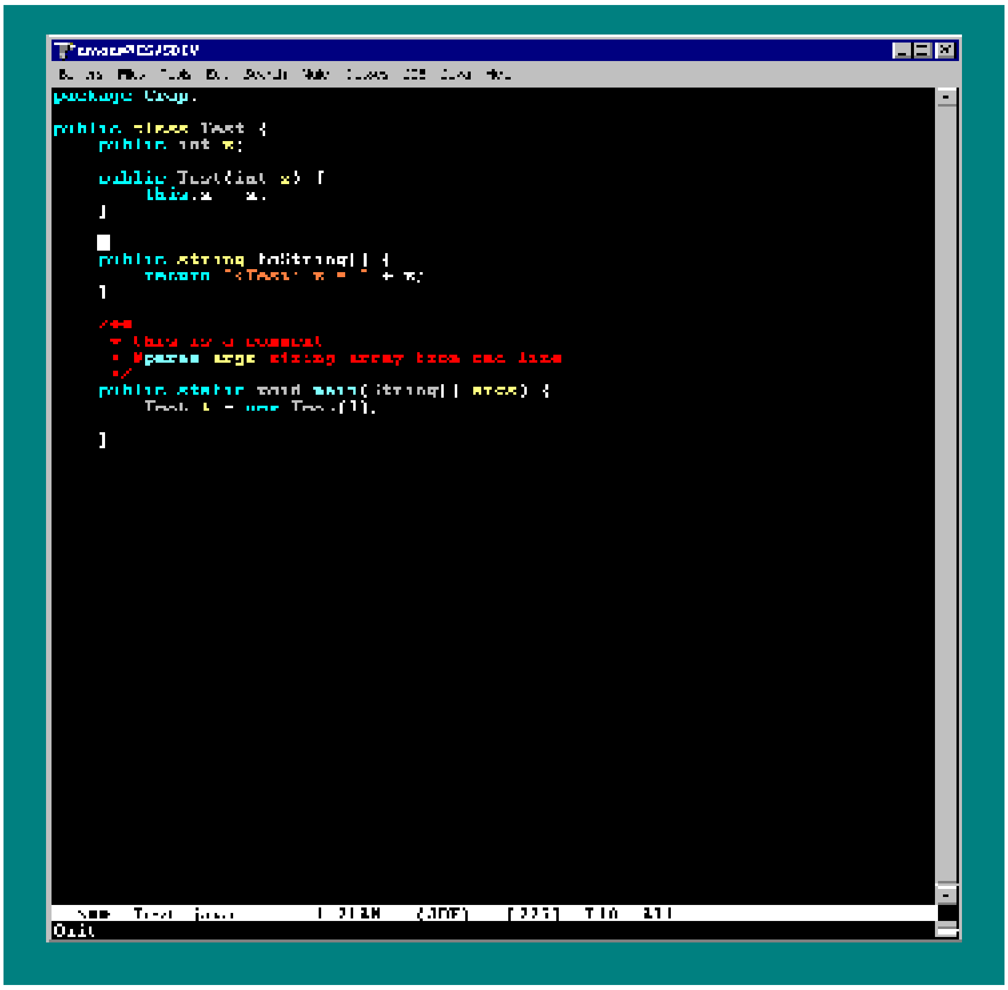
Visual Studio .NET is not the ideal teaching tool for C#. Students are not the primary audience for this product. Rather, the target is the professional programmer wanting to develop XML web services. The wizards included fail to aid in developing an understanding of what is happening. The configurability only gets in the way. The long list of features and license requirements create additional hurdles for a student just trying to get started. Visual Studio .NET is too complicated and for many users too expensive for teaching C#. Tools like this take time away from teaching language and programming concepts and require an instructor to spend time familiarizing students with the development environment [17].

2.2 Open Source

The field of Open Source IDEs available for .NET development is relatively small. The prevailing theme among these free editors is configurability and extensibility. Yet once again, these features advanced users demand fail to address the needs of a novice.

There are several general purpose text editors. One of the more popular is emacs [19]. Like many general purpose editors, emacs is highly configurable. In fact, there is a programming language called emacs-lisp designed specifically to configure emacs. Using this language, emacs can be expanded to handle the editing of virtually any programming language. In fact, it has been extended to support C# [22], but the use and even the setting up of these tools for use with C# takes some advanced knowledge. The default interface for emacs is also the extreme opposite of what Visual Studio .NET offers. Instead of the extra windows, emacs has very few, if any windows to clutter the desktop, as can be seen in Figure 2.2. With little exception, open source editors are not designed specifically for C#. They all have little tricks and shortcuts to make programming easier, but learning all of them can take considerable time and detract from the main goal of learning programming concepts.

There is one editor, however, that is trying to target C# specifically. SharpDevelop [12] is an open source IDE that is trying to clone Visual Studio .NET. It is a fairly well developed and organized project. It provides a decent free alternative to Visual Studio .NET. However, its likeness to Visual Studio .NET also makes it an imperfect solution for education. In almost every aspect, SharpDevelop tries to mimic Visual Studio .NET and so has the same complicated interface and levels of abstraction that can lead to confusion. The text editor though is has some features



```
package [illegible].

public class Test {
    public int x;

    public Test(int x) {
        this.x = x;
    }

    public string toString() {
        return "<Test: x = " + x;
    }

    /**
     * This is a comment
     * @param args string array from cmd line
     */
    public static void main(string[] args) {
        Test t = new Test(1);

    }
```

Figure 2.2 : Emacs user interface

worth exploiting. Because of this, we decided to utilize the SharpDevelop code base in the development of Dr. C#. This area is covered in depth later.

So, looking at open source editors, we find that while the monetary problem is solved, complexities still exist. These free editors are developed by advanced programmers for use by other advanced users. Programmers tend to grow accustomed to a certain interface and enjoy the ability to keep that interface as they move to new languages or tasks. If it is important enough to them, they will learn how to configure their editors to handle each new task. One can expect beginners to not have peculiar tendencies requiring configurability and so an editor with a simple intuitive interface is desired and preferred.

2.3 Interactivity

One key feature missing from the world of C# development is interactivity. This is one feature is a great aid for students trying to understand what their code is doing. The traditional development process is a continuous cycle of editing, compiling, and then running the code. There are several reasons why this seemingly innocuous process can be both time consuming and discouraging to a student.

A one line edit to a file can trigger a time consuming compilation. If all one wants to do is test out a simple method in their program, one must go through the hassle of creating a class that contains a main method, that has the appropriate references, and that has appropriate and explicit output statements. The student is forced to anticipate what output he wants. To make one change or to see the value of a variable or some other information, the edit-compile-run cycle must be repeated.

In addition, even the conceptual overhead of using the main() method is large. Students are left to wonder what keywords like $public$, $static$, and $void$ mean when they make their first program. Also, there is the issue of passing command line parameters, array access, and other explicit I/O considerations.

When a student needs to setup some context in which to test his code, often times a student will modify the main() method. In the cases when the test fails, the student must revert to a previous version, but often it is difficult to imagine what the previous state is. All too often, even to advanced developers, a small change will cause bugs and will require a rollback, but because the original source code was changed, reverting back becomes very difficult. The ability to test, play, and interact with code without the dangers of upsetting working code can aid both novice and advanced programmers.

2.4 Addressing the Need

Dr. C# is conceived to address all these issues. Constant attention is paid to making sure the student user focuses his time on learning programming concepts. To avoid presenting confusing windows to the user, the interface has been considerably simplified. Superfluous configurability options are missing. Wizards are absent. Dr. C# is a small download and is free and licensed under the GNU General Public License [6].

The most important feature added to the world of C# development is the addition of interactivity. To enable this feature, we have created a C# interpreter and have integrated it into an editor as a REPL. As a result, the old edit-compile-run cycle is replaced with an edit-compile-run-run-run. . . cycle which is far more convenient and flexible. The REPL breaks out of the confines of the main() method and allows the user to type in any valid C# statement or expression and have it immediately evaluated and its result printed to the screen. If the user forgets to test something or print out a variable for inspection, the user simply types in the commands to do so without having to recompile his code. Both students and advanced users can take advantage of this powerful new capability.

Chapter 3

Related Work

Since C# is still a very young language, the body of work relating to it is relatively small. However, the idea of an interactive programming environment has been around for a while. Dr. C# is heavily influenced by Dr. Java [26], which in turn is influenced by Dr. Scheme [18]. There is also substantial work in the open source community to develop tools for the .NET platform. The most notable of which is the Mono project [10] and the Shared Source CLI developed by Microsoft [9].

3.1 Dr. Java

As mentioned earlier, the predecessor to Dr. C# is Dr. Java. Dr. Java is an IDE for Java also developed at Rice University. One of the goals of the Dr. C# project is to recreate the Dr. Java look and feel for the C# world. As a result, the interface of Dr. C# has a similar layout including a simplified text editor and more importantly an interactive window with a REPL.

While Dr. Java serves as a model for Dr. C#, the implementations of the two projects approach the problem from opposite ends. The Dr. Java interface consists of two main parts, a text editor and an interactive window, known as the interactions pane, where users can type Java commands and have them interpreted immediately [15]. The engine behind the interactions pane is a program called DynamicJava [21]. This is a very powerful program which supports the entirety of the Java language, including the definition of new types. This is done through Java's reflection capabilities. Dr. Java is coded entirely in Java partly as a way to demonstrate the power and flexibility of the Java programming language. Similarly, Dr. C# is coded entirely in C# to showcase the language and utilize the features of the .NET Framework.

Most of the work for Dr. Java involves building an easy to use text editor and integrating DynamicJava as the engine for interactivity. The development of Dr. C# however has no interpreter to rely on. Rather, there is a relatively advanced C# editor in SharpDevelop. As such, instead of developing an editor and integrating with an outside interpreter, as in Dr. Java, Dr. C# starts at the other end of the spectrum, with an editor and has developed and integrated a C# interpreter. As such, many of the development issues encountered during Dr. Java development are not the same as those faced when developing Dr. C#. In the end, however, they both arrive at the same place and represent interactive pedagogic programming environments.

3.2 Dr. Scheme

Dr. Scheme [18] is the predecessor to Dr. Java. It is also a complete IDE with an integrated text editor, debugger, and REPL interactions pane. It is used widely in academic institutions for teaching introductory computer science classes and is also used by more advanced users.

3.3 Mono

The Mono project [10] is an attempt to produce an open source implementation of the Microsoft .NET Framework. In particular, they are trying to implement the Common Language Interface (CLI) standard of the .NET Framework as well as a standardized C# compiler [5]. The Mono project is a large and organized project and has had much success. They share the same goal of producing free tools for .NET development, and it is worthy to note that Mono is developed completely in C# just like Dr. C#.

3.4 Shared Source

Microsoft took the CLI and developed a version it released under its Shared Source license called the SSCLI [9]. This license is a Microsoft attempt at open source.

One of the reasons they produced this project is to essentially be a proof of concept that the CLI is a real standard and is implementable on other platforms besides Windows. The SSCLI is implemented for the FreeBSD operating system and includes a working compiler and .NET runtime. While we would have liked to have taken parts of this code for Dr. C#, the SSCLI is implemented in C++. As such, this does not fit into our goal of having a system written completely in C#.

Chapter 4

The REPL

The majority of the effort to develop Dr. C# goes into making the REPL. The engine behind the REPL is a C# interpreter. At the time of this writing, no other C# interpreters are known to be under development.

The main features of the REPL are as follows:

- a quick almost instant response to the user
- the ability to evaluate any valid C# expression or statement
- easy integration into a development environment
- dynamic loading and unloading of user created assemblies
- automatic printing out of any evaluation results
- user friendliness

This chapter will explain the development process of and then dive deeply into the architecture of the REPL explaining how the features are met.

4.1 Development

Development of Dr. C# is done on a Microsoft Windows 2000 .NET Server with the .NET Framework version 1.0 installed. Programming is done on Visual Studio .NET with source control provided by Visual Source Safe 6.0.

The most important feature of Dr. C# is its interactions window. Since, the development of Dr. C# started with the objective of recreating Dr. Java for the C# world, the process starts with a search for equivalent Dr. Java components for C#, most notably a brother to DynamicJava. This avenue does not prove very fruitful, and so other efforts are made to port code from Dr. Java to C#. This also is not successful. Subsequently, after exploring various parts of the Mono project, a decision is made that the REPL has to be made from scratch with a little help from Mono tools.

4.1.1 Looking for DynamicC#

Dr. Java development greatly benefits from having DynamicJava already developed. So, an effort is made to see what has already been developed in the area of C# hoping that there might be an equivalent of DynamicJava for C#. Unfortunately, there is nothing even close to the power DynamicJava could provide.

The supposed C# interpreters that are available can hardly be called interpreters. They simply take the C# statement, concatenate it with some code that wraps it in a class with a main method, and then dynamically compiles the code and executes it. This process proves both slow and cumbersome for various reasons.

First of all, it is impractical and costly to have to wait for the whole compilation process to simply print out something like a string or an integer value. Unlike the Java Virtual Machine which has the ability to interpret code, code in the .NET runtime is never interpreted. The process of running code in the .NET runtime can be seen in Figure 4.1. At compilation time, the code is not actually compiled to native machine code. Rather, .NET code is transformed into the Microsoft Intermediate Language (MSIL), and upon first execution, the code is Just-In-Time (JIT) compiled into native code. Even though the delay is small, it is definitely noticeable. This sort of behavior runs contrary to the interactive and responsive interface we are trying to achieve.

Additionally, these programs have no idea of keeping an environment. Since there is no parsing of the input, any notion of environment is nonexistent. The only way to achieve some semblance of an environment is to save the previously typed input and constantly feed that into the compiler. However, this is not a viable solution either. Often the user wants to print out the value of various things. If the environment is maintained in this way, each time the user presses enter, every previous output statement is executed again. This will quickly get annoying.

Finally, these systems demand strict adherence to C# syntax. Since these programs simply dump code into a main method, they can only accept C# statements. Often, the user would prefer to simply enter an expression like a variable name, but unfortunately, an expression cannot appear in the middle of a block.

4.1.2 Dynamic Compilation

Another idea to meet our requirements utilizes the ability in the .NET Framework to dynamically define classes, add methods to them, and execute code. This ability creates what is called an in-memory assembly and utilizes the libraries that can emit code at runtime. This is primarily designed for use by people who need to create classes dynamically at runtime. A substantial amount of work would be required to produce a REPL based on this kind of execution engine, but this idea became feasible after examining code from the Mono project. In particular, we made an attempt to leverage their C# compiler which had all the code necessary to emit MSIL. There are a couple implications had the REPL gone this route.

First, the fact that the program would have a parser gives the REPL some idea of what the user is typing. After the user makes some input the parser will generate an abstract syntax tree (AST). This is very important because at this point, the REPL is allowed to do some processing on this object before it emits code or does any kind of interpretation. As such, the REPL can know when there was a variable declaration or some other statement. Also, the REPL gains the ability to distinguish between statements and expressions. The idea of an environment now becomes possible since the input is not just blindly passed to a compiler.

Secondly, the Mono parser parses the entire C# language. Utilizing this parser, the REPL gains the ability to not just parse statements and expressions, but also new type declarations like classes and delegates. The REPL would be much more powerful than originally envisioned. In fact, it would essentially gain the same functionality as DynamicJava.

In the end, the decision is made to not to use the entirety of the Mono parser and compiler. It simply offers too much for the purposes of the project. As an alpha release, it is decided that the ability to support the defining of new types is unnecessary. Of course, one can simply get around this by reworking the parser to not accept new type definitions, but more importantly, the process of emitting code relies on having the code JIT compiled before it executes. Once again, the delay introduced by this step is unacceptable for a responsive and interactive REPL. However, dynamic type creation can be leveraged in the future to extend the capabilities of the REPL.

4.1.3 Porting DynamicJava

In a last ditch effort to avoid having to create a C# interpreter from scratch, the decision is made to try and use a Java to C# conversion tool. At the time, the only freely available conversion tool is provided by Microsoft. It is called the Java Language Conversion Assistant (JLCA) [7]. This tool comes with promises of easy conversions between the languages; however, upon first use one quickly realizes that while there may not be any errors that occur during conversion, the resulting code produced is not error free.

The handling of converting Java features like anonymous inner classes is not handled very elegantly. The task is further complicated by the fact that the mapping of Java classes to their equivalent C# class is not one to one or many times there is not even an equivalent class. For example, C# does not have a LinkedList object. Rather, it only has an ArrayList type of list which does not offer the exact same functionality. As a result, the resulting code is hard to understand and does not always function as originally advertised. It does not take long to decide this effort should be abandoned.

The search for a DynamicC# fails to materialize. As a result, the decision is made to create a C# interpreter from scratch. There are some advantages to this approach. Unlike the case with Dr. Java, development of the Dr. C# REPL will be controlled locally and so bug fixes and extensions are more easily controlled.

Additionally, the REPL represents a test of the power of the C# language and how far the .NET Framework can be pushed.

4.2 REPL Architecture

The REPL essentially has three main components. The parser is the first component that requires a lot of attention. Once an AST is generated from the parser, the interpretation is accomplished by utilizing the visitor design pattern. Finally, to add the ability to load and unload user classes dynamically, the REPL takes advantage of a .NET component called the AppDomain.

4.2.1 Parsing

Parsing for the REPL is not as easy as one would assume. The task is made harder by the requirement that the user be able to input any valid statement *or* expression. The C# grammar originally only lets expressions appear in the context of some statement. For example given the variable declaration statement of $intx;$ and the expression x, a typical C# parser would complain if it just found the expression in the middle of a method body. However, to make things as easy as possible for the user, the REPL should be able to accept and process the simple expression.

As such, the grammar for the parser is not simply the C# grammar as defined in the C# language specification [5]. It is modified to accept both statements and expressions. This causes problems because ambiguities arise and limits the types of parser techniques available to tackle this problem. The grammar is essentially right recursive with an infinite look ahead.

Recursive descent parsing

The first attempt tries to create a recursive descent parser. This proves to be very difficult however as recursive descent parsing requires a left recursive grammar. An attempt is made at transforming the C# grammar, and we try to use some recursive descent parser generators written in C# like CoCo/R [2]. These tools did not work as the lookahead proved to be too difficult for them to handle. Subsequently, a hand written parser from scratch is attempted but once again proves futile.

Lex and yacc

In the interest of time, the decision is made to utilize a C# port of lex and yacc. This lexer and parser generator pair is popular as a tool for building bottom-up table driven parsers. This particular port is produced by the makers of Mono and is used to produce their C# parser for their compiler. The grammar fed as input to the parser generator also originally comes from Mono, but significant modifications are made. In particular, the Mono parser begins to build an environment and provide some context for their next compilation step. This code has to be removed. The only remnants of parse-time processing is some code that transforms primitive short hand type references like int to full name references like System.Int32. Additionally, modifications to the grammar are made to accommodate the input of both statements, expressions, and also using declarations. Using declarations are used to declare namespaces to be used in the REPL.

AST generation

The result of parsing is an AST. There is an object to represent each possible statement or expression construct in the C# language. To make the AST structure flexible, Statement and Expression objects inherit from the abstract ASTNode class and are designed with the visitor design pattern. There are three main visitor interfaces IExpressionVisitor, IStatementVisitor, and IASTVisitor. The first two are self explanatory. The IASTVisitor implements the other two visitors and provides a way to make a single visitor be able to traverse every ASTNode.

An AST intermediate representation is important for the REPL. It allows for some sort of processing or preparation before the user's input is actually interpreted. By using the visitor pattern, the AST gains even more extensibility. For example, the ToString() method is implemented as a visitor. There is also the possibility of having a visitor for type checking. Other possible features are discussed later in this thesis.

4.2.2 Interpretation

The most important object in the REPL is the InterpretationVisitor. Just like the name says, it performs the actual interpretation of the user input. There are several supporting classes that handle type information and assembly loading that take advantage of the advanced reflection and runtime type management capabilities of the .NET Framework. These classes work with a combination of the visitor and state design patterns to make interpretation possible. To make user code reloadable, the interpreter utilizes the .NET Framework ability to isolate the environment in which code executes.

Type management

The most important supporting class is the TypeManager. It handles both user and system assembly management, namespace management, and class name to actual type resolution.

In Java, the root of reflection operations and access to classes and interfaces at runtime is the Class type. In C#, the equivalent is the Type type. According to the C# API, Type is the primary way to access metadata and is the gateway to the reflection API [1].

Typically, when a C# program is compiled, the code is automatically packaged into what is known as an assembly. An assembly can be thought of like a Java jar file in that it is a container for compiled classes, but it has the added ability of being an executable. As such, assemblies come with the dll and exe filename extensions.

Whenever a type is referenced in the REPL, it will eventually go though the TypeManager. This includes both primitive, library, and user types. This means that before any user input can be interpreted, the TypeManager must be initialized and have loaded any of the types that may be used. Thus, upon initialization, the TypeManager loads system types that are commonly used. Those types are located in System.dll and mscorlib.dll.

Assembly loading The are several steps to loading an assembly into the REPL. First, system assemblies are assumed to be installed into a standard location called the global assembly cache (GAC). The GAC is a special directory for assemblies that are frequently used. Assemblies in the GAC have been though the JIT compiler and are native code. When the .NET runtime is asked to load an assembly, by default, the GAC will be searched if a specific location for the assembly is not specified. When an assembly created by a user is loaded, the full path of the assembly needs to be specified.

It is not enough just for an assembly to be loaded into memory for the REPL to gain access to the types contained within. The REPL has no concept of what types belong to the system or to the user in the .NET runtime in which it finds itself. The TypeManager must query each assembly to discover what types are available. Luckily, an Assembly object is created when an assembly is loaded and provides methods that make type extraction easy. During this process, a Type object for each type is discovered from an assembly and cached into a hashtable that is keyed by the fullname property of the type.

At the same time a type is discovered from an assembly, the TypeManager is able to determine what namespace a particular type belongs. A namespace in C# is akin to a Java package. They can be nested within each other and any classes created within a namespace may be referenced by its fullname, which includes the namespace. There is also the ability to import namespaces into a source file. In C#, importing is accomplished by the $using$ statement. For example, in Java one might see $importjava.util.*;$ while in C# one will see $usingSystem.Collections;$.

When a type is extracted, the TypeManager treats system and user assemblies slightly differently. In both cases, given a type like System.Collection.ArrayList, the System and System.Collections namespaces are added to a table of valid namespaces. The difference between system and user assemblies is that the REPL assumes that the user will want use classes from his assembly and so automatically does the equivalent of creating $using$ statements for any user defined namespace. Namespaces found in a system assembly must still be explicitly added to the environment with a $using$ statement.

Another slight difference is that system assemblies are automatically loaded when the user declares a $using$ statement for them. For example, if the user enters $usingSystem.Windows.Forms;$ into the REPL, the System.Windows.Forms.dll file will automatically be loaded and its types and namespaces registered in the environment.

The reason for automatic user namespace loading and system assembly loading is to make things easier for the student user. The first feature is intuitive for this kind of environment. The user will want to use his own classes in the REPL and having to explicitly declare such and action is just unnecessary work. The second feature comes about as a result of dealing with Visual Studio .NET and its reference mechanism. In Visual Studio .NET a user must explicitly add a reference to a library even besides just adding a $using$ statement. This extra step is really unnecessary as the IDE should know where to find the particular dll the user is trying to reference. The REPL recognizes this and so tries to make things easier for the user.

Class name to type resolution Once the classes are registered, class names can be translated to their actual Type objects. Some sort of mechanism is necessary because there is no way to simply ask the .NET runtime to find a class by handing it a string. There is another caching mechanism used in this process that keeps track of the short hand names, instead of fully namespace qualified names, of types. This cache works in conjunction with the namespaces that are currently in use and maps short names to the fully qualified names. For example, if the user declares that he is using the System.Collections namespace. He can declare an ArrayList object in the REPL as opposed to having to declare the full name of System.Collections.ArrayList.

The whole process of name resolution is relatively straightforward. The method that performs the resolution takes in a string representing the type to lookup. It will first check the short name cache to see if it was previously used. If so, it will have the fully qualified name. If not, the method will assume that a fully qualified name was given. At this point, the method performs a lookup in the table of registered types to quickly return the Type object desired. If the lookup fails, there are two possibilities, either the string passed in is a short name and is not previously used or the type has not been registered with the TypeManager. In the first case, the desired type name is concatenated to each entry in the list of currently used namespaces and a type lookup is performed. If a lookup is successful, the short name is cached and the Type object is returned. If the lookup fails, then the only possibility left is that the type has not been registered with the REPL and an exception is thrown.

A TypeManager object is associated with each InterpretationVisitor instance. It is not only used for class name resolution, but can also be used for type checking and related tasks like making sure operators are used correctly.

Visitors and States

The InterpretationVisitor is the most important class in the REPL. All classes in some way or another are designed to support this class. This class works by visiting an AST and traversing the tree performing the actual interpretation of the input. In the course of interpretation, this visitor must go through different states. Instead of keeping flags around for this, the state design pattern is utilized in combination with the visitor pattern. The interpretation process depends very heavily on the reflection capabilities in the .NET Framework.

The InterpretationVisitor is instantiated with a TypeManager and an Env environment object. The environment will persist in the interpreter until the REPL is reset. As such, the InterpretationVisitor cannot be treated like a singleton.

Since the IASTVisitor interface requires so many methods to be implemented, an abstract class called AASTVisitor with default implementations of all the interface methods is provided. This seemingly trivial class makes the job of developing other visitors much easier. A couple of these visitors will soon be discussed.

The best way to describe the InterpretationVisitor is to detail the steps it takes to interpret various types of input. Of course, the visitor follows the steps detailed in the C# Language Specification [3], but the details in which the specification is implemented are worth discussing.

Whenever possible any exceptions generated by the interpreter during evaluation are returned to the user with helpful messages. In most cases the messages will be the same as those the C# compiler or runtime will return when it finds an error. For example, the user may mistype a class or variable name. This will generate a message asking the user to check that he is not missing any assembly references, just like in Visual Studio .NET. This is done in to make any transition both to and from Visual Studio .NET as easy as possible.

The visitor itself has no default entry point. After an AST is generated, it can accept the visitor. There are several cases of interest in how the visitor works: simple mathematical computations, binary operations and boxing, statement and expression transformations, loops, member and element access, and assignments.

Before diving into these cases, one important question needs to be answered. That question is *What is the going to be the result of executing the visitor?*

An important conclusion made in the middle of the development process is that the result of any user input is either going to be some computation which returns some value or some kind of variable property or variable in the REPL environment. The visitor methods are designed with this in mind. They will either return the computed value of the user's input or make a lookup into the REPL environment and return the associated property or variable value.

To facilitate the passing of information between the different methods of the visitor, the Info class is created to act as a wrapper for computed information. For example, variables in the REPL environment are stored as VariableInfo objects which provide a way to associate a variable name, the associated object, and the declared type. The other Info objects used will be described in the following case discussions.

Mathematical computations For simple mathematical computations, the visitor does not rely on any reflection. Rather it simply performs the raw computation required. For the simple case of the user's typing in of $3 + 4$, the parser will return an AST that is a Binary object with a left and right property being IntLit objects which are integer literals. The forBinary() method essentially will evaluate the left side and the right side of the AST, getting integer results back. It can then perform the desired operation and return the result.

Binary operations and boxing The Binary object is of particular interest because C# supports operator overloading and the concept of boxing and unboxing. The REPL does not support user defined operator overloading currently, but must support the overloaded operators defined in the C# specification. In particular, the + is overloaded for $string$ concatenations. The + and the - operators are also overloaded to handle delegate hooking and unhooking respectively. Delegates are a feature of C# that can be thought of like C function pointers. However, they are first class types.

Boxing is a special feature of C# that Java lacks. Boxing allows primitive types to be treated as objects. For example, it is perfectly acceptable to have a statement like $3.GetType();$ or $object i = 2;$. This feature causes some special problems when it comes to interpretation involving binary math operators. Once again using the example of $3 + 4$, the method initially only knows that the result of the left and right sides of the AST are of type $object$.

The problem is that binary operators like +, -, *, and others are overloaded for each particular primitive type, but not just each type, but pairs of each type. In order for a binary operation to succeed, the .NET runtime must be able to make some determinations as to what the types being operated on are. For example, the + operator, with regards to primitive types, is only defined for use with two int, uint, long, ulong, float, double, or decimal types. Note, this means, when the runtime encounters a expression like $1.1 + 2$, which tries to add an int and a float, there is no + operation to handle it and some processing must be done before a result can be obtained.

This processing is called numerical promotion. It is a set of rules that defines how primitive types can be converted to other types in order to fit one of the signatures of the operators. The promotions are done in a way such that information is not lost, hence the term promotion and not demotion. Float types are promoted to double when necessary and not vice versa.

Since the interpreter essentially is the runtime of the REPL, it must handle such type differences itself. If the .NET runtime had one int and one $object$, it could determine that some unboxing would need to happen with the $object$ type and perform the requested operation because it is given some context. Simply given two object types, however, it cannot add or perform other binary operations because there is no context for which it can determine which particular overloaded binary operation to use. There is no $+$ operation defined for $object$ types. It is this problem that the makes the binary expression a nontrivial expression to interpret.

The interpretation of a binary expression must perform the numerical promotion itself by querying the for the types of the left and right hand expressions. Furthermore, it must select the appropriate overloaded binary method to use. Additionally, the REPL cannot simply cast the left or right value to any primitive type, in particular as a way to get around having to implement numerical promotion, or a casting exception will be thrown. For example, a boxed float cannot be cast to a double even though there is no data lost. The upshot of this is that each primitive type and operator must be checked against every other primitive type. This causes the number of methods for this interpretation to multiply very quickly.

The problem of not having a context in which to unbox primitives arises also in dealing with unary operators. In this case, however, each case can simply be enumerated as opposed to having to multiply the cases.

Statement and expression transformations The next case of interest is how the interpreter performs some translations to certain statements and expressions in order to simplify and reuse code defined for other operations. For example, the compound assignment statement $x+=3;$ is simply shorthand for normal assignment statement $x = x + 3;$. There is no reason the interpreter should write extra code to handle the shorthand. So, the interpreter will perform a simple transformation from any compound assignment to its respective normal assignment statement. This type of transformation is repeated for operations like unary mutators, i.e. $x++$ becomes $x = x + 1$.

Loops The next construct needing special design considerations are loops. Loops pose a special issue because they contain a state. In the interpretation of a loop, it is necessary to keep tabs on the test statement which determines whether to continue with the loop or leave. This naturally creates two states, in the loop and not in the loop. A straightforward approach to this is to use a binary flag in the main visitor, but loops present more problems than just two states. Special control flow statements like $return$, $break$, and $continue$ have to handled somehow. It quickly becomes apparent that a more elegant solution is needed.

The answer is the a combination of the visitor and state patterns. A visitor for the loop state is created to handle the special information needed to properly interpret loops. The LoopVisitor extends the InterpretationVisitor because it needs only to override the methods dealing with loops and loop control flow.

Some of the special handling provided in the LoopVisitor has to deal with the scoping of for and $foreach$ statements. The for statement has to have a scope for the variable declaration, test statement, and iterator statements. The $foreach$ statement has to have an enumerated object in its scope. Supporting methods are needed to check to ensure proper functionality.

In addition, there is the issue of control flow. In the course of interpreting the body of a loop, one may encounter a $return$, $break$, or $continue$ statement. Control flow poses a special problem because it is not simply a matter of state. It makes no sense to be in a $break$ state. When one of these statements are encountered, it essentially causes the current sequence of operations to cease and return control to a previous context. In this respect, these statements are much like exceptions. As such, the interpretation of these control flow statements results in exceptions that must be handled by the methods handling loops. For example, if the interpreter is evaluating a while loop and encounters a $break$ during interpretation of the body of the while loop, a BreakException is thrown. The forWhile method is required to catch this kind of exception and deal with it accordingly.

Passing information and .NET reflection Up to this point, the interpreter has not needed any of the reflection capabilities of the .NET Framework. The need arises though when it comes to interpreting expressions like member access, element access, and invocations.

A member access as defined in the grammar is essentially an expression followed by a . followed by an identifier. The identifier is the access of some member of the expression defined on the left side. This expression poses an interesting problem in that evaluating the left hand side to its final value is not always desired. The left hand side can resolve to either a type, an object, or variable, but sometimes there could be a casting operation of some sort which has some information that may affect how the member access should be evaluated. As a result, the interpreter needs a way to encapsulate the result of evaluating the left hand side as well as the other pertinent information.

As mentioned earlier, this is where the Info objects come into play. Besides VariableInfo objects, there are also ArrayInfo, NamespaceInfo, TypeInfo, and ObjectInfo objects. ArrayInfo objects contain the array and along with its indexers. NamespaceInfo and TypeInfo contains strings and Type objects respectively. The ObjectInfo is the most interesting of the bunch. It contains not just the object currently being evaluated, but also the current type of the object (current, because it could be cast to another type), the reflection MemberInfo that is used to access the current object, as well as the original type and object used to object the current object.

These Info objects cannot be the final result of any interpretation, with the exception of the VariableInfo. They are used internally by the interpreter to pass information as part of the runtime interpretation of user input. This runtime processing is another state for the interpreter to be in. This particular state has use in assignments and so is incidentally called the LValueVisitor. The term lvalue is appropriate in this sense because a member access has a left hand side that needs to be evaluated but is not the final result of a statement evaluation.

At this point, a quick discussion of the .NET Framework reflection capabilities will aid the forthcoming examples. As mentioned earlier, the Type object is the gateway to the reflection API. A Type object may be asked various properties about a particular type, like whether it is abstract, an array, an interface, a value type, etc. The most important method is GetMember() as it will return MemberInfo objects. These objects are the heart of the reflection API. They represent the building blocks of a class, i.e. constructors, properties, methods, fields, and events. The only way to access or invoke these building blocks is to find the corresponding MemberInfo object.

The interpreter relies heavily on these objects. A few examples will illustrate this more clearly. The first will cover a member access, followed by an invocation, and finally assignments.

Member access Take, for instance, the member access $Console.Out.NewLine$. This member access gets the current newline character of the output stream from the $Console$ object. Figure 4.2 shows what the AST looks like.

From the top of the tree, the visitor method for member access will first evaluate the left hand expression with the LValueVisitor. This visitor will recur to the bottom of the tree where there is a SimpleName object representing the $Console$ type. A SimpleName as an lvalue can be one of a variable, a namespace, or a type. In this case the method returns a TypeInfo containing a Type object for the $Console$ class. At this point, the visitor follows the rules defined in the C# language specification with regards to member access evaluation. If the left hand of the member access is a type, then the interpreter must check whether the given identifier in the member access is another type or a member of the returned type. In this case, reflection determines that Out is a static property of the $Console$ and gets the PropertyInfo representing that property. The LValueVisitor returns a ObjectInfo object with all this discovered information back to the original InterpretationVisitor. With all this information the interpreter must once again go through the steps described in the language specification to determine whether the result is a type, an indexer access, a method group, property, or field access. After performing the necessary checking is the interpreter able to determine that the string $NewLine$ refers to a property of the TextWriter Out. The interpreter makes a reflective call for that property and returns the result.

Method invocation Method invocation is very similar to member access but has two added issues, one being overloaded method names and the other being delegate invocation. For example, Console.Out.WriteLine() has several overloads. The left hand side of a invocation must evaluate to either a ObjectInfo or a local variable. In the first case, all the information needed to perform the reflective call to invoke the method is passed back when the left side is evaluated. While there is a complicated set of rules for method overload resolution. The .NET reflection API has greatly simplified this task by providing a way to let the library do all the resolution. One simply uses the Type object for a class and calls InvokeMember passing in the appropriate flags and parameters for method invocation.

Delegate invocation is also easy. Once the parameters for the method call have been evaluated, delegate classes have the ability to be dynamically invoked by passing in the parameters in an array.

Assignments Assignments in the interpreter present another special challenge when it comes to type checking. The LValueVisitor plays an important role as one would expect. An assignment consists of two parts, an lvalue and an rvalue. An lvalue is property or variable that can be assigned to. The rvalue is some object that be assigned from.

In type safe languages like Java and C#, it is important that assignments are always of a valid type. For example, a $object$ type can never be assigned to a int because there is no way to implicitly convert between the two classes. The C# compiler enforces such rules at compile time. However, the REPL does not perform any type checking when it comes to assignments. In the case of an assignment to a locally defined variable in the environment, the assignment will always succeed. If the user later uses the variable incorrectly, an exception will be thrown by the runtime claiming types are incorrect. If the user tries to make an assignment to some property and the types do not check, the runtime will enforce type safety and will complain about the operation, but it is important to remember that these type checking errors are generated by the runtime and not the REPL.

Putting it all together

The sheer number of classes and methods available for use in the interpreter can be intimidating to someone trying to programmatically use the interpreter. Fortunately, all the basic functionality for communicating with the interpreter is isolated in a single interface, IInterpreter, which is implemented by the InterpreterProxy class. The interface defines the following important methods:

- string Interpret(string);
- void SetUserAssemblyPath(string);
- void LoadUserAssembly(string);
- void LoadSystemAssembly(string);
- void SetStdOut(TextWriter);
- void Reinitialize();

The first method is the most important. The string parameter passed in is the input from the user and the return value is going to be the result of evaluating the input string. The first step in this process is that a parser is created and handed the input. The generated AST then executes the InterpretationVisitor which will return either null, a variable in the environment, or some object. Before returning to the caller, the result is always transformed into a string by having its ToString() method called.

One may argue that the result of evaluation will not always be a string, but more times than not, the user is going to use the REPL to check the value of some property or expect some sort of string to confirm some action has taken place. By automatically calling the ToString() method on the result, the user is free from having to always make the call himself.

The methods dealing with assemblies control which and when types are loaded into the interpreter runtime. These methods allow the user to specify exactly where to load an assembly from.

Because the interpreter will be integrated into a window in Dr. C#, output from a user's program needs to make it to this window. The SetStdOut method allows for this.

Finally, the reinitialize method resets the environment and causes the TypeManager to reload to its original state which does not have user code loaded.

Notice the details of setting up the TypeManager, the Env, the InterpretationVisitor, parsing, and so on are abstracted away to make programmatic use of the interpreter as easy as possible.

Implementation notes

In the course implementing this C# interpreter, the language specification is followed as closely as possible. However, due to project goals and peculiarities in the .NET Framework, some deviations from the specification are made.

Unsafe code C# is designed with many features from C, C++, and Java. It has many of the desirable features from Java that has made Java a popular language for teaching. At the same however, C# has the capability to perform operations from the C and C++ world that many instructors find undesirable. A lot of these language features fall under the term *unsafe code*. Since Dr. C# is designed to be an editor for beginning students, we deemed the support of unsafe code unnecessary.

User defined conversions C# also supports user defined conversion operations. In Java and the .NET runtime, implicit conversions like upcasting are done automatically. Even explicit conversions through casting only succeed if there is a direct inheritance chain between and object and the desired cast. User defined conversions, however, can convert one type to another completely unrelated type. User defined conversions can be either implicit or explicit, but either way, will usually cause confusion for the person reading the code. Thus, this feature is also not supported.

Multidimensional arrays In the area of arrays, C# supports multidimensional arrays. In the course of implementing support for arrays, it is discovered that given an array of type int[][,] the reflected type will produce an array of type int[,][]. This seems to be a bug in the .NET Framework and is reported to Microsoft. It does not affect the performance of the interpreter however.

Boxing As mentioned earlier C# supports boxing of primitive types. However, boxed primitives are not considered first class objects. In particular, take the following variable declarations:

```
int a = 3;
int b = 3;
object c = 3;
object d = 3;
```

c and d are declared as $object$ types but are assigned to ints. C# automatically boxes the ints so the assignment operations succeed. If a user wanted to compare a and b for equality, he simply types $a == b$ and gets true. However, if a user tries to compare c and d for equality by typing $c == d$, the result will be false. Intuitively, this is somewhat wrong. The == operator is defined for int objects. Asking c for its type also reveals that it is an int. The object oriented programming mindset would say that the result of the == operation should have been true, but this is not how boxing is implemented in the .NET runtime.

Because the interpreter must mimic the .NET runtime exactly, it must also return false in this circumstance. However, the task is not very straightforward. In the implementation of the == binary operation, because of boxing, the result of evaluating the left and right operands comes back as object types. The interpreter must make a query to ask whether the operands are primitive types in order to use == correctly. However, in the myriad of reflective properties one can discover about an object, there is no way to determine if an object is a boxed primitive. Asking a and c for their types returns the same thing. This piece of missing information causes the interpreter to have to include special code to handle this case. In particular, the interpreter must always ask for the declared type of the operands in the == operation, as opposed to asking the operand itself for its type, which can differ from the declared type.

Operator overloading C# allows operator overloading. This feature is a remnant from C and C++ and is not supported in Java. Since our goal is to create a copy of Dr. Java, this feature is also not currently supported in the interpreter.

Taking advantage of .NET

The .NET runtime is also called the Common Language Runtime (CLR). Microsoft builds .NET on the philosophy that one should program in the language that best suits a particular task. The CLR is designed to run code from any language as long as it is compiled for the CLR. It is not just the ability to run code from multiple languages though. It is also possible to make cross language calls. For example, a C# program can use a library written in Visual Basic .NET, Scheme, or any other language that will run in the CLR. Dr. C# gets to take advantage of this for free by being built on the .NET Framework. The TypeManager can load any dll or exe for .NET. As such, it can load libraries written in other languages. As long as the user is inputting C#, the user can use the REPL to perform cross language evaluation.

4.2.3 Application domains

One of the most important innovations in the REPL is its ability to dynamically load and unload user types without having to perform any special operations with operating system processes or modifying the runtime environment. The operating environment of the REPL is different from most other runtimes. A typical runtime gets a block of compiled code with an entry point and just starts to execute the code. Along the way it may determine that certain classes may need to be loaded into the runtime. The nice thing about these classes is that the runtime knows that the compiled assemblies will not change during execution time. In fact, to ensure this, the runtime will acquire a write lock on an assembly. If the user edits his code and recompiles, the user has to stop the runtime to release the lock first and then restart the runtime so it can reload the new assemblies. This programming process is inconsistent with what the REPL is tasked with.

Programming using a REPL

The REPL for Dr. C# expects the user to make changes to his code in the middle of execution and recompile. The first issue is not how user classes can be dynamically loaded, assemblies are loaded dynamically all the time. The issue is that once those classes change, how can the REPL reflect those changes because there is typically no way to unload a type from the runtime. Additionally, the user's code typically does not have just one particular entry point. He can call any function anywhere, anytime he wants. There are infinite entry points. The REPL must not only be able to provide a quick response, but also must make sure not to write lock any of the user's files and ensure any changes to an assembly are reflected in the runtime of the REPL before the user enters his next input for interpretation. Since there is no single entry point, the REPL must make sure any and all types the user might reference are loaded into memory.

In Dr. Java, this behavior is achieved by restarting the REPL process each time the user compiles his code. This scheme is not perfect. Because communication with the REPL is done though Remote Method Invocation (RMI), Dr. Java has run into synchronization issues that have to be worked around. Additionally, there is overhead associated with starting and stopping processes and each time this is done there is a chance a process may get lost.

.NET provides an easy mechanism for getting this functionality with a construct called an application domain, which is represented by the AppDomain object. An application domain is essentially an isolated environment where applications can execute. According to the API, application domains:

> provide isolation, unloading, and security boundaries for executing managed code. Multiple application domains can run in a single process; however, there is not a one-to-one correlation between application domains and threads. Several threads can belong to a single application domain, and while a given thread is not confined to a single application domain, at any given time, a thread executes in a single application domain [1].

It may be easier to think of an application domain as kind of like a process. Each one has their own environment and set of security rules governing what code it is allowed to load and execute. A process though can have multiple AppDomain objects at the same time however. They always at least have one as the startup application domain. AppDomains can communicate with each other as well, not just across the same process or even between processes. They can span networks as well. An application domain can exist on any computer.

AppDomains in the REPL

The REPL makes use of two application domains. The first one is created by the .NET runtime upon initialization and has the assembly containing the interpreter's classes loaded. The main class in this domain is the InterpreterFrontEnd. This class also implements IInterpreter. Its initialization process however is different from the InterpreterProxy, which is the main class of the second application domain.

While the InterpreterProxy sets up a TypeManager and other supporting classes in its constructor, the InterpreterFrontEnd needs to create the proper environment for the rest of the interpreter to run. Once an AppDomain is created, its properties cannot be changed. The original application domain has certain properties that are not consistent with the behavior needed, hence a second application domain is necessary.

In particular there is a property called ShadowCopyFiles that, by default, is false but needs to be set to true. This property determines whether assemblies loaded into the AppDomain are to be shadow copied. Shadow copying an assembly means that before being loaded, an assembly is copied into a temporary directory first and then loaded. This way, the system does not obtain a write lock on the original assembly. This is very important to the REPL as it makes the assembly free to the user to recompile his code without having to worry about stopping the interpreter from running. Other properties set by the InterpreterFrontEnd tell the second AppDomain where to look for assemblies by default.

The separation created by the second application domain also gives the desired features mentioned earlier regarding loading and unloading types. Once a type is loaded into an application domain, it cannot be unloaded from memory. The only way to unload them is to unload the entire application domain they are initialized in. Figure 4.3 shows the REPL architecture using the AppDomains and a rough flow of events for an interpretation call.

.NET remoting The InterpreterFrontEnd creates and communicates with an instance of InterpreterProxy in the second domain. This communication is an application of .NET remoting, which is the .NET version of RMI. The code to create this is very simple. In fact, the details are automatically handled by the .NET runtime leaving only one line needed to set everything up:

```
InterpreterProxy ip = (InterpreterProxy)userDomain.CreateInstanceAndUnwrap(
“DrcsInterpreter”, “Rice.Drcsharp.Interpreter.InterpreterProxy”);
```

The $userDomain$ is the application domain where the user’s code is loaded and executed. The method $CreateInstanceAndUnwrap$ is short hand for two function calls, $CreateInstanceFrom$ and $ObjectHandle.Unwrap$. The former takes two parameters, the name of an assembly file and the name of a class to be loaded from the given assembly, respectively. It returns an object handle to the created instance in the other AppDomain. In order for the object to be used in the local AppDomain, it needs to be unwrapped, which is what the second method does.

This unwrapped object has two special properties. One is that the type it represents does not need to be loaded into the local application domain. Second, the unwrapped object is called a proxy and all communication between the proxy and the real object is handled automatically by the .NET runtime. These proxy objects make programming very easy because they are treated just as if they were local objects.

At the low level, .NET remoting communicates by using the Simple Object Access Protocol (SOAP). SOAP is described as a lightweight XML based protocol designed to exchange structured and typed messages over the web. It the underlying protocol for many other web service protocols.

Since communication is designed for the web, not every object can be communicated between application domains. In order for an object to be used in this scenario, it must either be serializable or a MarshalByRefObject. A serialized object is copied and transported between AppDomains. MarshalByRefObject is the base class for objects that communicate across application domain boundaries by exchanging messages using a proxy. Both InterpreterFrontEnd and InterpreterProxy extend MarshalByRefObject.

4.2.4 Summary

The interpreter back end of the REPL forms the basis of an interactive interface into the C# language and is the main feature of Dr. C#. At the beginning of development, public projects did not suit the needs of the REPL, and so it is developed from scratch. The parsing uniquely deals with both statements and expressions in generating an AST, even handling the ambiguities that occur when statements and expressions are considered the same level of language construct. The interpretation of the AST is architected around the visitor and state design patterns. Type management is handled carefully to make things easy on the user. The interpretation classes are presented programmatically though a simple interface and takes advantage of the .NET framework's ability to isolate and execute code.

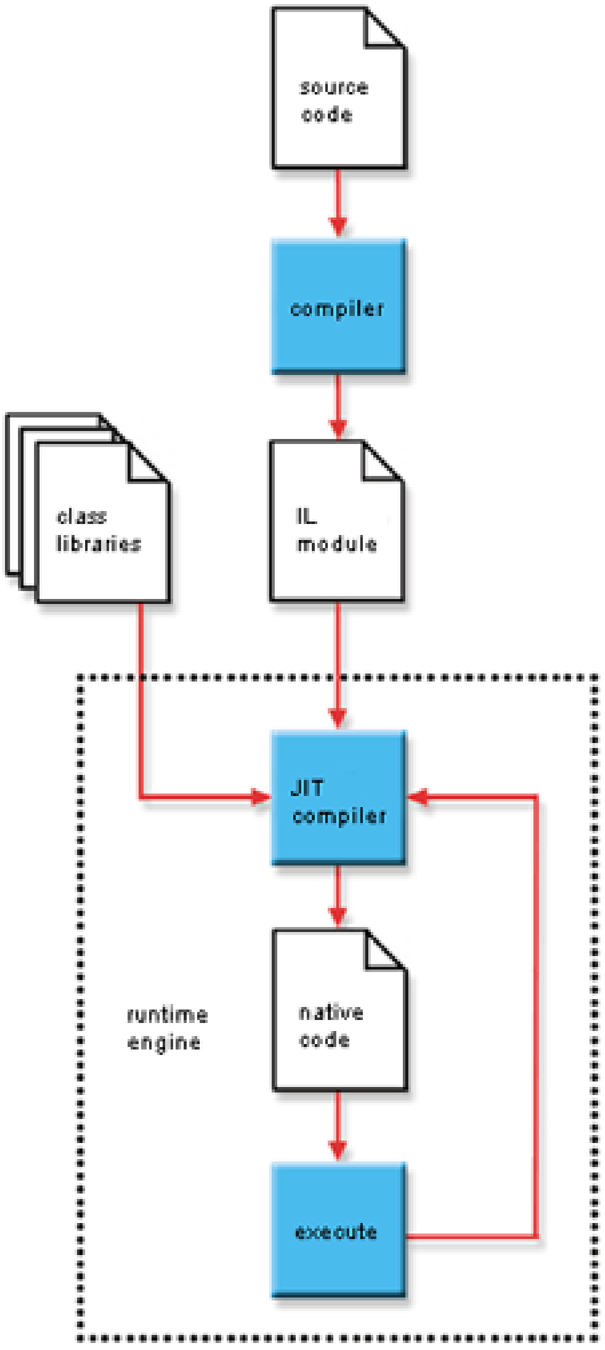


Figure 4.1 : .NET compilation and execution process

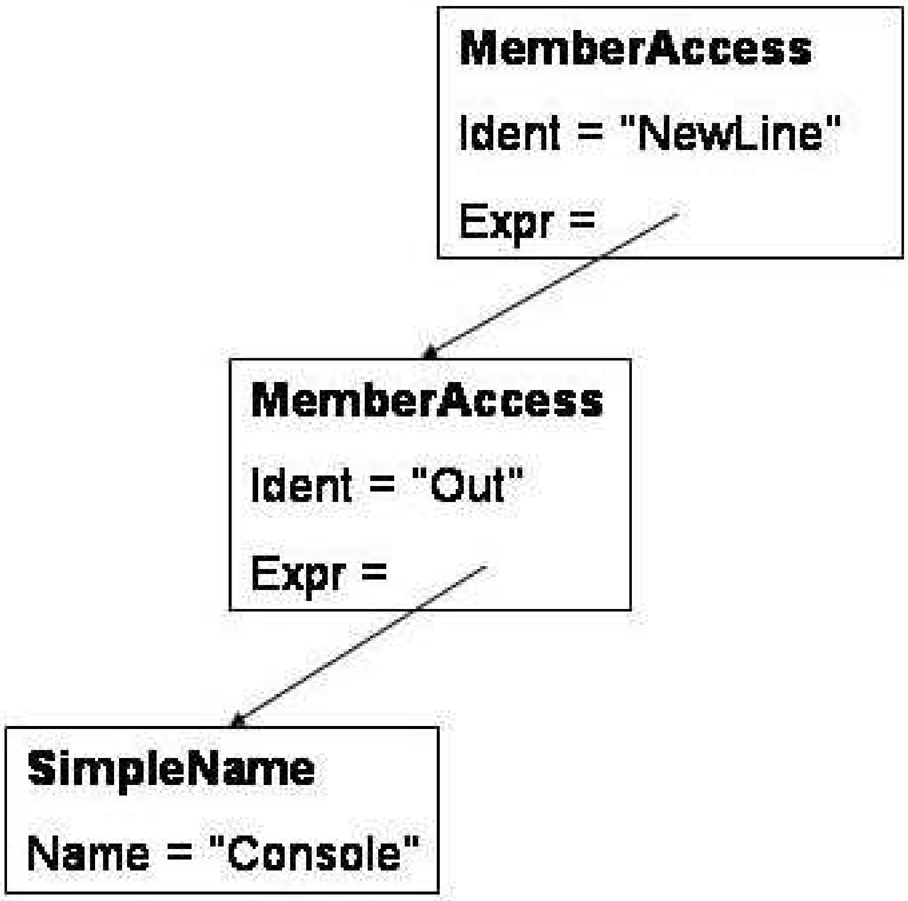


Figure 4.2 : AST of the member access Console.Out.NewLine

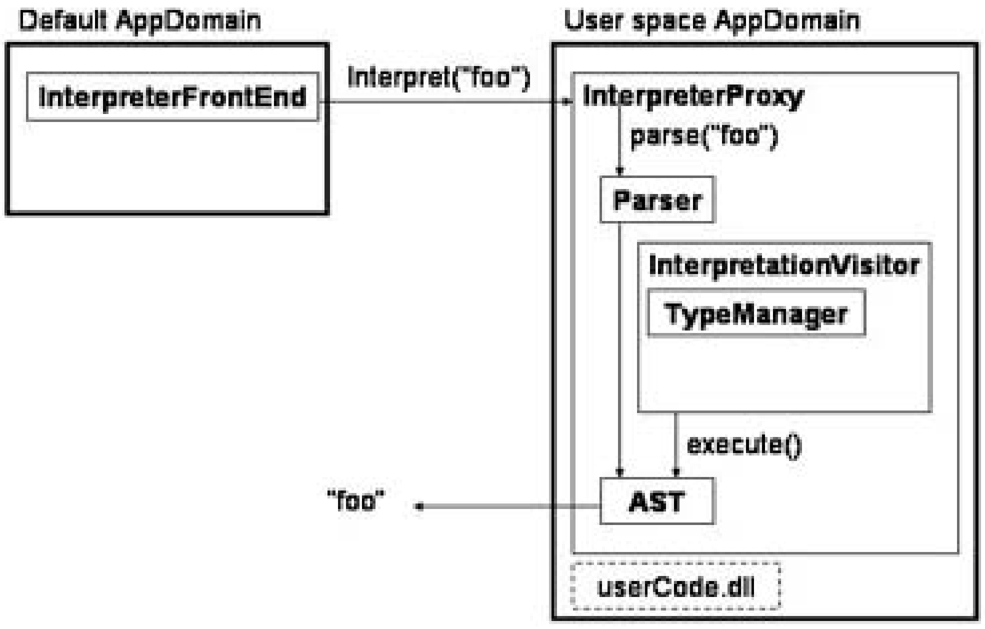


Figure 4.3 : REPL architecture and execution flow

Chapter 5

The Editor

The editor for Dr. C# is built on top of the SharpDevelop (#D) development environment [12]. The decision to build on top of an existing open source project came after several options are considered. One of the goals of Dr. C# is to provide an easy to use interface to the student, free of superfluous buttons and functions, taking the Dr. Java interface as a model.

5.1 Searching for an editor

The editor needs to handle basic conveniences like syntax highlighting, indentation, and other little features that make coding easier for the beginning student. At the same time, the editor should be advanced enough to satisfy more seasoned programmers who want to take advantage of the interactivity provided by Dr. C#. The first idea is to utilize Visual Studio .NET itself. The second option is to build one from scratch. The decision in the end is to utilize an open source editor designed for use with C# as a basis for our own editor.

5.1.1 Visual Studio .NET

The dominant IDE for C# development currently is Microsoft Visual Studio .NET. As mentioned earlier, this program is developed to be the most powerful tool for use by software professionals in the production of XML web services. To cater to the large spectrum of needs of the professional, Visual Studio .NET is feature rich and includes wizards to try and simplify many aspects of development. However, the needs of the professional and the needs of the student are very different. As mentioned earlier, the Visual Studio .NET interface is not well suited for a beginning student, but we thought perhaps Visual Studio .NET could be simplified to suit our goals.

Visual Studio .NET has an AddIn architecture that allows third parties to create separate applications and have them tightly integrated into the main IDE. In fact, the Visual Studio .NET Integration Program (VSIP)[13] will make available certain APIs not generally available to the public.

If we are to utilize the VSIP, a substantial effort would be put into disabling many of the advanced features of Visual Studio .NET. Dr. C# would not be a stand alone program but would become a toolbar among all the other toolbars and a window in the IDE no different from all the other windows already there.

It did not take too long to decide that this is not the path we should take. The REPL needs to integrate with a simple text editor which provides simple conveniences to the programmer, not a full fledged IDE like Visual Studio .NET. Integrating into Visual Studio .NET also means that any distribution of Dr. C# has to be accompanied by a large download and potentially expensive license tied to the main editor. One goal for Dr. C# is to be a free and lightweight alternative for students. Additionally, we would be tied to an API that could potentially change. As such, this idea is abandoned.

5.1.2 Building from scratch

Building an editor from scratch is not too daunting of a task. In fact, the editor for Dr. Java is available for porting to C#. Using the same JLCA that is used in an attempt to port DynamicJava, we made an effort to do just that. Unfortunately, the results are the same as with the earlier attempt. The mapping between Swing classes and the Windows.Forms classes is not one to one. The number of modifications needed to properly fix the code generated by the JLCA is simply not worth the time. Even after the substantial amount of effort needed to make the code compile, the code would still be hard for a developer to use because it is still automatically generated code stricken of comments and documentation.

5.1.3 SharpDevelop

As mentioned earlier, SharpDevelop is an open source editor that is attempting to make a clone of Visual Studio .NET. It offers many of the same features #D is well suited for our purposes for many reasons:

- licensed under the GPL
- good text editor featuring convenient features
- architected to allow addins
- active developer community to provide updates and testing
- written in C#

The most important feature of #D for the development of Dr. C# is that the source code is freely available for download and modification since it is distributed under the GPL. Without this, development of Dr. C# would be much more difficult. #D tries to clone not only many Visual Studio .NET features, but also its user interface. Once again, the target audience for this IDE is the advanced developer. In order to transform #D from its default interface as seen in Figure 5.1 into a suitable lightweight IDE for students that looks like Dr. Java as shown in Figure 5.2, significant modifications are made to the source code of #D. These changes are only made possible by the rights granted to us under the GPL.

#D has a good text editor providing many convenient features like syntax highlighting and code indention which advanced developers find essential and students find helpful. It is convenient that Dr. C# is able to benefit from a well developed text editor so time could be focused on other priorities.

Another very important feature is that #D is architected to allow third party code to integrate into the development environment as a addin. In fact, the interactions window is added to the environment through this feature. More about this is discussed in the next chapter.

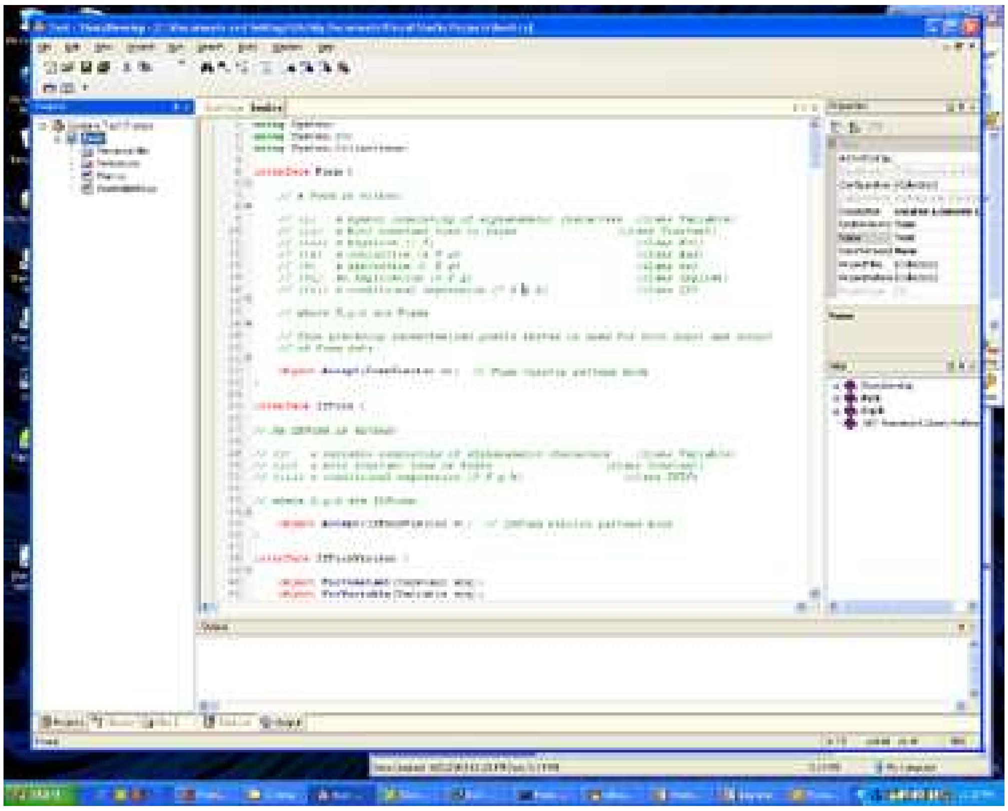


Figure 5.1 : SharpDevelop default user interface

The active developer community provides two advantages. One is that updates and new features are constantly being developed. If any of the new features added to later versions of #D are deemed convenient to have in Dr. C#, they can be added very easily. Additionally, #D has its own set of test code and so testing on Dr. C# can focus on the REPL and integration as opposed to the editor.

One of the goals of Dr. C# is to demonstrate the power of the C# language by developing the program in C# itself. As such, it is fortunate that the developers of #D have the same goal for their IDE. Being written in C# allows for easy integration of the REPL into the IDE. The features in the .NET framework that let the REPL communicate internally are used again for communication between the IDE and the REPL.

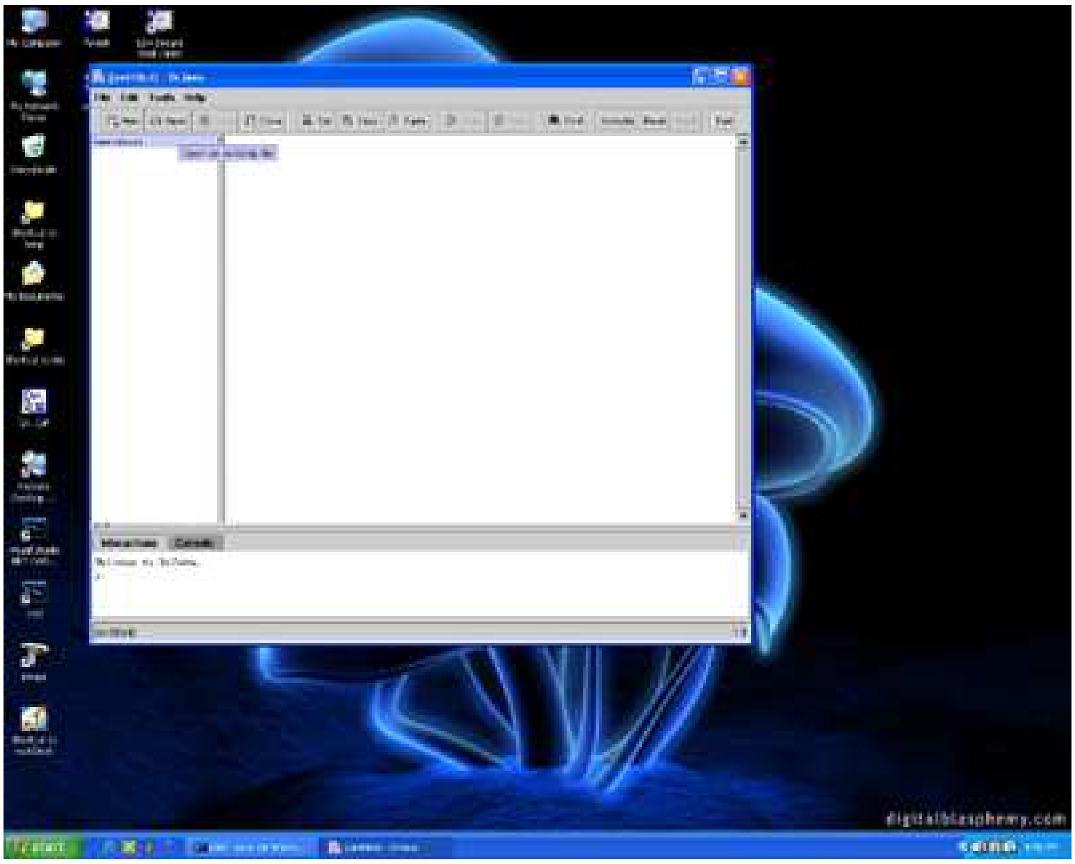
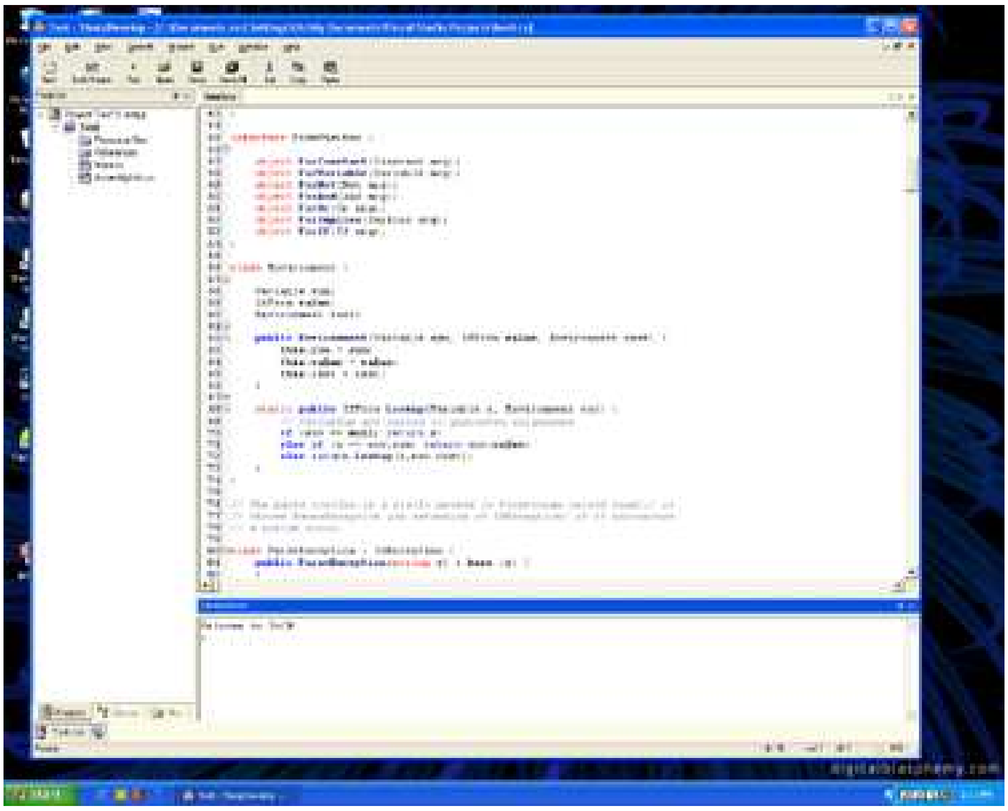


Figure 5.2 : Dr. Java user interface

5.2 Modifying SharpDevelop

Various modifications are made to SharpDevelop to make it suitable for our target audience. A snapshot of #D is taken at the .89 beta release for our modification. The changes made to the source tree are marked in the source code for future modification. The majority of changes made are to simplify the interface. In the end, the interface can be seen in Figure 5.3.

Many methods originally designed to return toolbars or start other third party applications are disabled. The number of configuration options has also been substantially reduced. The default toolbar is changed to reflect the same options as Dr. Java. A couple interfaces are added to allow the IDE to integrate with the REPL. One defines how a class in a process can let itself be killed by a remote process.



```
[illegible]
```

Figure 5.3 : Dr. C# user interface

The second interface provides a way to make sure remote services are kept alive. The need for both abilities will be discussed in the next chapter.

The project structure for files edited in #D is also kept. This structure allows the user to organize his code in a logical fashion. It is important to have some way to define all the classes and references necessary for a application to work because every compilation in C# requires the generation of an assembly file containing all the files in a project. In Java, it is possible to recompile one class in a project while leaving other class files alone, but that compilation model does not translate to C#. It is awkward to have an assembly for each class.

Chapter 6

Dr. C#

The combination of the editor and the REPL is what makes Dr. C# a uniquely integrated development environment. The REPL is an addin in #D and manifests itself as a dockable window in the user interface. The editor and REPL run in separate processes following the model of Dr. Java. Development of the REPL and editor proceed independently of each other for the majority of the development time. When the two are put together, new bugs and hurdles have to be overcome.

6.1 AddIns

The REPL actually integrates with the editor though the SharpDevelop addin architecture. The addin compiles into a dll (DrCSPlugin.dll) that gets loaded dynamically. Upon initialization, #D checks its AddIns directory for any files with a .addin extension and automatically loads them. The interactions window is defined as a pad within #D. This way, it can either be free-floating or docked and integrates seamlessly into the developing environment like any other window.

The addin extends the AbstractPadContent class provided by #D. To do so, it overrides the $getControl$ property and makes it return the panel that contains all the other GUI components we are using. SharpDevelop uses this property to add the pad in the right places. By overriding the $getIcon$ property, the addin is able to display its own icon.

The panel contains a component of type InteractionsBox. This is a specialized version of a TextBox designed for our purposes. It allows the user to edit only the last line. Text that gets inserted at the end appears before the last line where input occurs. It also holds a history of the last 20 lines that were entered and can recall them when the user presses the up or down keys.

The constructor of the InteractionsPlugin class first calls the base constructor and tells it that the panel is to be named “Interactions”. It then proceeds to add the components to the panel and register itself with SharpDevelop’s ServiceManager. This needs to be done in order to kill the addin at the end of the program, which is done manually since a circular relationship prevents the garbage collector from automatically doing this. It also makes it possible to call certain functions from toolbar button handlers.

In the constructor, the addin also registers two event handlers for SharpDevelop’s “start build” and “end build” events. When a build is completed successfully, the REPL will reload the newly generated assembly. The addin also initializes the remoting part of the addin, which will be covered in detail later.

The addin provides simple functions to clear the interactions window, to add text to its bottom, and to interpret user input. The InteractionsBox instance actually calls the interpreter function directly if the enter key is pressed.

The interpreter function first checks for two commands that can be handled locally (“_clear” and “_restart”) and processes them if applicable; otherwise, it uses remoting to asynchronously let the Dr. C# interpreter process the input. The asynchronous call allows the editor to remain interactive while the user input is evaluated.

Getting the REPL to evaluate user input is not as simple as passing it a string. One important aspect of the architecture of the REPL functionality now comes into play. The interpreter for Dr. C# and the editor run in separate processes. The next section will discuss the reasons for this.

The addin architecture makes future additions to the IDE much more of a possibility. Possible additions are discussed in the Future Work chapter.

6.2 Process separation

The editor and the REPL run in separate processes in the .NET runtime. This design is done for mostly the same reasons as it is done in Dr. Java. Running in the same process poses several problems:

- a malicious or just unknowing user can manipulate the editor or REPL execution
- user code can call Application.Exit() and end the Dr. C# process
- user code alters the Dr. C# environment
- user code can enter an infinite loop or perform some other intensive task causing Dr. C# to become unresponsive

The first problem occurs if a user asks the current application domain for access to classes it has already loaded. The REPL has no idea what code the user asks it run and it is perfectly conceivable that a user could knowingly or unknowingly alter the editor. By separating the REPL and the editor processes, there is at least some measure of protection from accidents.

The second issue is more typical. If Dr. C# ran in one process, then if somewhere in the user's application there is a call to end the current application, that call would also end Dr. C# itself. The call to exit is not always obvious when it happens and so this can lead to confusion and lost data. The easiest way to prevent this is to have the editor and REPL run separately since the call to exit can be buried deep in the user's code and there is no easy way for the REPL to prevent the call.

Users may also need to alter the runtime environment in which their code runs. Dr. C# has to have its own runtime settings as well. Separation once again allows both the freedom to control their settings.

The last reason can often not be predicted to happen but certainly is an undesirable state to be in. Students often write code that has bugs like infinite loops that chew up the processor and leaves the editor slow or unresponsive. Granted, the REPL could simply always run user code in a separate thread, but stopping a thread in the middle of execution can often lead to an unstable state in the runtime. A cleaner way to deal with this problem is let user code run in another process, and if problems arise, just kill the separate process.

6.2.1 The implementation

The addin code creates the REPL process upon initialization. The separate process is called DrCSServer. When this process initializes, it creates an instance of InterpreterFrontEnd and registers it as a remotable service.

Communication between the two processes once again uses .NET remoting. In this case, a little more setting up has to take place than communication between the application domains in the REPL. The two processes pass information though two TCP channels that are currently fixed at port 8086 (from interpreter to addin) and at 8085 (from addin to interpreter). If one of these two channels is already open, the addin and interpreter will not be able to communicate. A more robust way needs to be developed in the future to handle this case.

In the constructor, the addin first opens the incoming channel and exposes the interface of the InteractionsWriter class through it. This makes the InteractionsWriter accessible from the outside. Its function is to receive text output from the interpreter and relay it into the addin's InteractionsBox. To do this, it is given a pointer to the InteractionsPlugin so it can use the methods to append text.

After the incoming channel is opened, the interpreter is spawned. This is done in the InteractionsPlugin.GetRemoteInterpreter() method. It first checks if the remote process is still running; if it is not, it needs to be restored. The main Dr. C# process checks to make sure the interpreter is running by essentially "pinging" the interpreter. If the ping ever fails, for example, if the user makes a call the Application.Exit() or if the user has to explicitly kill the REPL as a result of an infinite loop, the main process will have to create a new process and initialize everything again.

Using Process.Start(), the interpreter, which resides in its own executable, is started. On the interpreter side, the outgoing channel is opened and the interface of the InterpreterFrontEnd class is exposed, making it accessible to the addin. The main thread of the interpreter then proceeds to sleep indefinitely.

Back in the addin, GetRemoteInterpreter() checks if the interpreter process was spawned successfully. If it was, the process ID is stored; if not, an exception is thrown. After waiting a second, the addin checks if the interpreter process that is just spawned is still running by checking its exit code. If it has quit, then it is possible that another process of the interpreter is still running, which prevents the new process from creating the channel. In this case, the addin searches for the interpreter and tries to bind to the old instance.

Once a running interpreter is found, the addin connects to the outgoing channel that is created in the remote process. It does so by requesting an InterpreterFrontEnd instance from the channel and storing it in a local variable. All functions specified in this interface are now available from within the addin as if it is a local object.

The addin now uses the newly acquired interface to supply the Interaction-sWriter to the interpreter. The InteractionsWriter is a TextWriter and becomes the stdout and stderr of the interpreter. As a result, when there are any calls to Console.Out, the interpreter writes to this writer, and the text will appear in the interactions panel of the editor.

Finally, GetRemoteInterpreter() starts the pinger thread again. This pinger is an instance of the InteractionKeepAlive class and simply executes the interpreter's PingServer() method every 10 seconds. This causes the interpreter to write an empty string to the InteractionsWriter, which has no visual effect but completes the circle. This way, both channels are exercised regularly and thus kept from closing.

The function that interprets user input uses a delegate to enable asynchronous execution, but the fact that remoting is used is totally transparent at this point. When the delegate call return, the returned value is sent to the InteractionsBox in the main editor window.

When SharpDevelop is shut down, the SharpDevelopMain.Main() method asks the ServiceManager for the instance of the pinger thread and stops it. Using the same means, it will ask for the plugin's instance and execute its KillServer() method to manually kill the interpreter process.

6.3 Bugs and testing

The integration of the editor and REPL causes several difficult to solve bugs. Finding a solution to them is often difficult as the architecture is pushing the limits of what the .NET Framework is designed to do. Assistance is offered by program managers and developers at Microsoft, but sometimes even they could not solve the problem.

6.3.1 Bugs

The first bug we encounter happens because we did not use the two application domain architecture for the REPL. After the user completes a build, the REPL will automatically load the assembly into memory, but this causes it to be locked. When the user tries to rebuild, the locked assembly could not be replaced. Fortunately, the use of shadow copying in the separate application domains solves this problem.

Another turns out to be what we believe is a bug in TextBox class in the .NET Framework. When text is sent to the InteractionsBox, for display, the AppendText method is called. The problem we found is that if the InteractionsBox is not showing and the AppendText method is called, an ArgumentOutOfRange exception is thrown. This exception would cause the entire Dr. C# process to freeze. Only after a lot of digging did we determine that instead of calling AppendText, we can simply concatenate to the Text property of the InteractionsBox to print before the window is displayed. This seems a little like a hack, but it works. The AppendText() method must be performing some undocumented operations.

The pinging mechanism implemented may seem somewhat unorthodox, but that scheme evolves from a tendency for the main process to lose communication with the REPL process after a period of inactive time. We discovered that sending messages back and forth between the two processes will keep the two alive and get rid of the problem.

Other problems are more annoyances than bugs, Visual Studio .NET often locks files and assemblies for no reason. The only way we get around this is closing and reopening the editor.

6.3.2 Testing

Testing is made easier by having an extensive tracing mechanism based on XML debugging output. It is possible to turn off and on debugging output for most methods and classes. Finding particular output is made easier by making sure the output conforms to XML by having an open tag when some method or area is entered and a closing tag when that same method or area is entered.

Also an effort is made to try and produce unit tests for the REPL and integration using NUnit [11]. Much of the interpreter is tested, but many more cases need to be produced before Dr. C# can move beyond a beta version.

Chapter 7

Future Work

The version of Dr. C# being released is an alpha version. There is certainly much room to grow. Fortunately, the editor is continuously being updated by the developers at SharpDevelop. At some point, we may want to break totally free of the dependency on them and develop the editor in a similar fashion as Dr. Java. The REPL will need to be updated as the C# language evolves. The next version of the language will have new features that will need support.

7.1 Changes to the REPL

Since this is the first version of the REPL, there are many possible additions. In particular, the interpreter has the ability to be extended to be as powerful as DynamicJava and to support runtime type generation. While this feature is not supported, the reflection features of the .NET runtime makes this a possibility.

Work began on making the interpreter type check, but since DynamicJava also does not do type checking, the visitor to perform this task is not fully developed. This is certainly one feature that could be useful and not too difficult to implement.

The next version of C# promises to support anonymous delegates and generic code. These new features will also test the reflection capabilities of the next version of the .NET Framework. It remains to be seen how cleanly these features will be supported, but at the very least significant modifications would have to be done to the parser and interpretation engine of the REPL.

A feature of Dr. Scheme is the ability to have language levels. Dr. Java promises to have this feature as well. The addition of this feature to the REPL should not be too different from how it is added to Dr. Java.

One interesting extension could take advantage of remoting capabilities provided in the .NET Framework. There is nothing to keep the interpreter behind the interactions window from running on another computer somewhere. Since messages between application domains are by default transported using a protocol designed for the web, it is a natural extension to utilize this capability for perhaps some kind of distributed computing scenario.

7.2 Changes to SharpDevelop

A snapshot of SharpDevelop is taken as the basis for the editor in Dr. C#. Since then, #D has had a number of features added to it. The most notable, of which, is code completion. Code completion is a feature of Visual Studio .NET that many developers find very handy. It can be both a good and bad feature for beginners. It can help speed up development by skipping having to look things up in the API. The ability to look up and understand documentation though is a good skill to teach to a student. Code completion is not always going to be available.

Other areas of improvement are in development and can be added on a case by case basis.

7.3 Changes to Dr. C#

Dr. C# has much room to grow. At the very least, it can catch up to the Dr. Java in features. In particular, a C# debugger can be integrated. Since the REPL runs in a separate process, a debugger can set breakpoints in the REPL process and step through its code. This allows the user to invoke the debugger from any (non-private) point of entry, rather than solely through the main() method.

Another enhancement would be the ability to use the REPL in the context of a breakpoint environment. This allows the user to examine variables, perform method calls, and even change values before resuming execution. The capabilities provided in the .NET Framework reflection API could be leveraged for this feature. The Mono project is also developing debugger with hopes of this type of functionality and have contacted us about the possibility of using our code in their project.

NUnit can also be more tightly integrated in the future, as it is in Dr. Java. The addition of these features is eased by the addin architecture of #D. Dr. C# may become more suited for advanced developers though its ability to turn on and off different addins.

Another possibility for Dr. C# is the ability to run in the Mono .NET runtime. The Mono .NET Framework is currently incomplete, but SharpDevelop is capable of running in it. If the Mono runtime will properly support .NET remoting and application domains, then the REPL should work in Mono as well.

There is probably some room for improvement in opening the channels needed for editor REPL communication. Keeping the port numbers fixed can eventually lead to problems. Instead, the plugin should first determine two free ports and open the incoming channel on one of them. The other port number should be passed to the REPL, possibly via a command line parameter, and to be used for the incoming channel.

The development of Dr. C# will move away from being exclusively on Visual Studio .NET and Visual SourceSafe. The project will move to a model similar to how Dr. Java is developed by utilizing an open source editor and source control, namely SharpDevelop and SourceForge.

Chapter 8

Conclusions

Dr. C# starts as an effort to produce an IDE with a pedagogic focus. The development process starts with a survey of the state of available C# tools that are available for utilization. Perhaps because of the young state of C#, we fail to find many components worth capitalizing on for use in Dr. C#. We end up implementing a C# interpreter for use in a REPL and drastically simplifying an open source editor. The integration of the two results in the final product. Each stage of development presents its own unique set of bugs and challenges.

The REPL is capable of interpreting almost any valid C# statement or expression, with the exception of those we explicitly do not support like unsafe blocks. The REPL parser produces an AST that gives the interpreter a chance to do some processing on the input before it gets interpreted. The interpreter has its own type and assembly management separate from the .NET runtime. The interpretation of the AST is done with a mix of the visitor and state design patterns. The .NET Framework provides a way to separate types during runtime, and by utilizing this feature, the REPL gains the ability to dynamically load and unload user code.

The editor for Dr. C# takes it code base from the SharpDevelop project. The source is modified to be dramatically simpler and free of extra windows and functionality that can interfere with the learning process.

Integration of the two parts produces a full featured IDE that is simple yet powerful. The addition of interactivity to C# both helps beginning students and can also be utilized by more advanced programmers. The separation of processes allows the editor and REPL to exist separately without bothering each other. The end product meets the goals of being simple, responsive, user friendly, intuitive, free, and an example of the powerful capabilities of .NET.

Bibliography

[1] C# api. http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?cont%entid=28000519.

[2] Coco/r. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/CSharp/.

[3] Csharp language specification. http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpdownl%oad.asp.

[4] Defining the basic elements of .net. http://www.microsoft.com/\\net/basics/whatis.asp.

[5] Ecma standardization. http://msdn.microsoft.com/net/ecma/.

[6] Gpl. http://www.gnu.org/copyleft/gpl.html.

[7] Java language conversion assistant. http://msdn.microsoft.com/vstudio/downloads/tools/jlca/default.asp.

[8] Microsoft .net homepage. http://www.microsoft.com/net.

[9] Microsoft shared source. http://www.microsoft.com/\\resources/sharedsource/default.mspx.

[10] Mono. http://go-mono.org.

[11] Nunit. http://nunit.org/default.htm.

[12] Sharpdevelop. http://icsharpcode.net/OpenSource/SD/Default.aspx.

[13] Visual studio .net integration program. `http://msdn.microsoft.com/vstudio/vsip/default.asp`.

[14] Ben Albahari. A comparative overview of c#. `http://genamics.com/developer/csharp\_comparative.htm`.

[15] E. Allen, R. Cartwright, and B. Stoler. Drjava: A lightweight pedagogic environment for java. *SIGCSE 2002*, March 2002.

[16] David Berlind. Java vs. .net: How it's become a bet-the-ranch race. `http://www.zdnet.com/anchordesk/stories/story/0,10738,2830227,00.html`.

[17] S. Bloch. Scheme and java in the first year. *The Journal of Computing in Small Colleges*, 15(5):157–165, May 2000.

[18] R. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. Dr. scheme: A pedagogic programming environment for scheme. *International Symposium on Programming Lanugages*, pages 369–388, 1997.

[19] Free Software Foundation. Gnu emacs. `http://www.gnu.org/software/emacs/`.

[20] Ian Fried. $50 million from hp, microsoft for .net. `http://news.com.com/2100-1001-959066.html`.

[21] S. Hillion. Dynamicjava. `http://koala.ilog.fr/drjava`.

[22] Brad Merrill. Emacs tools. `http://www.cybercom.net/˜zbrad/DotNet/Emacs/`.

[23] Stuart Reges. Can c# replace java in cs1 and cs2? *ACM SIGCSE Bulletin*, 34(3):4–8, September 2002.

[24] Mike Ricciuti. Strategy: Blueprint shrouded in mystery. `http://news.com.com/2009-1001-274344.html`, October 2001.

[25] Carol Sliwa. Update: Microsoft gains visual studio .net momentum. http://www.computerworld.com/developmenttopics/development/java/story/0%,10801,78372,00.html.

[26] Brian Stoler. A framework for building pedagogic java programming environments. Masters, Rice University, April 2002.

[27] Alan Williamson. There may be trouble ahead. http://www.sys-con.com/java/article.cfm?id=1401.