Contract Wizard II

_____

Diploma Thesis

Dominik Wotruba

Supervised by Prof. Dr. Bertrand Meyer and Karine Arnout

ETH Zürich

October 6, 2003

Acknowledgements

Let me express my thanks to the people who made this diploma thesis possible.

First of all I want to thank Karine Arnout for her supervision, her scientific support and her very helpful comments on this diploma thesis.

I thank Prof. Dr. Bertrand Meyer for his consultations and supervision.

I would also like to express my thanks to Hans Dubach for his great help and support in dealing with administrational questions.

Last but not least I am very grateful to my parents for their support in every respect.

Abstract

This diploma thesis is part of a general effort to improve existing components by adding contracts a posteriori. In this diploma thesis, I focus on *.NET* components.

.NET does not support *Design by Contract*. For this reason, a tool called *Contract Wizard*, which allows adding contracts to every *.NET* assembly, whatever *.NET* language it was originally written in, was developed by Karine Arnout [2]. The *Contract Wizard* was written for a beta version of *.NET*. The goal of this project was to rewrite this tool for the latest version of *.NET*.

Contract Wizard II reads a "contract-less" *.NET* assembly and explicit contracts provided separately. The tool merges this information automatically, generates *Eiffel* proxy classes containing the contracts and compiles them using the *Eiffel* compiler into a new *.NET* assembly.

The tool provides incrementality: if the assembly already contains explicit contracts, the tool does not process the whole *.NET* assembly again, but takes advantage of an *XML* representation of the *Eiffel* proxy classes.

In this diploma thesis, I explain how to use the tool *Contract Wizard II*, I discuss its design and give a precise description of its implementation.

Content

Introduction

.NET does not support *Design by Contract*. For this reason, a tool called *Contract Wizard*, which allows adding contracts to every *.NET* assembly, whatever *.NET* language it was originally written in, was developed by Karine Arnout [2]. The *Contract Wizard* was written for a beta version of *.NET*.

The aim of this diploma thesis is to implement a tool called *Contract Wizard II,* which works with the latest version of *.NET*.

Contract Wizard II reads a "contract-less" *.NET* assembly and separately provided contracts. It merges this information and automatically generates *Eiffel* proxy classes containing contracts. Then the contracted *Eiffel* proxy classes are compiled into a new *.NET* assembly. Moreover, an *XML* representation of the contracted *Eiffel* proxy classes is generated. This enables the client to append contracts to an already contracted assembly without going through the whole process given above. For more details on the scope and the objectives of this diploma thesis, see *Chapter 1*.

Contract Wizard II has been designed with extendibility in mind: it should be easy to adapt it to support future changes of the *Eiffel* language and to automatic contract extraction from *.NET* assemblies [5].

This report consists of four major parts:
- The first part describes the project plan including the set-up of the project and the intended results.
- The second part is a user manual explaining how to use the tool *Contract Wizard II*.
- The third part is a developer report; it addresses the developers who want to extend the tool.
- The last part is theoretical and discusses the advantages and drawbacks of storing contracts directly in the assembly's metadata instead of storing them in separate *XML* files as implemented.

Part A: Project plan

1. Project plan

1.1 Project description

1.1.1 Overview

Building reliable software is the aim of software engineering. Assertions, which are part of the Eiffel programming language, provide the programmer with essential tools for expressing and validating correctness arguments. The key concept is Design by Contract. Viewing the relationship between a class and its clients as a formal agreement, expressing party's right and obligations, is the only way one can hope to attain a significant degree of trust in large software systems. [2]

The current implementation of *.NET* does not support *Design by Contract* yet. The goal of this project is to develop a tool, which provides the ability to add contracts like preconditions, postconditions and invariants to an arbitrary *.NET* assembly, which will be a big enrichment for many programming languages based on *.NET*.

1.1.2 Scope of the work

The main work of the diploma thesis is to implement a tool, called *Contract Wizard II*. There already exists a tool called *Contract Wizard* developed by Karine Arnout. The main difference between *Contract Wizard II* and *Contract Wizard* is the implementation.

I developed the entire tool *Contract Wizard II* in full *Eiffel for .NET* and take advantage of the *XML* technology (comparing to *Contract Wizard I* that is a mix of *Eiffel*, *C#* and *COM*). The current implementation (full *Eiffel for .NET*) makes it possible to turn *Contract Wizard II* into a *Web service* later.

I implemented the tool considering future *Eiffel* language extension, like contracts for attributes. Furthermore, all inserted contracts are stored in *XML*. This has the advantage that for example, tools for automatic contract extraction [5] can easily be incorporated.

The work consists of the following parts (see *Figure 1* on next page):
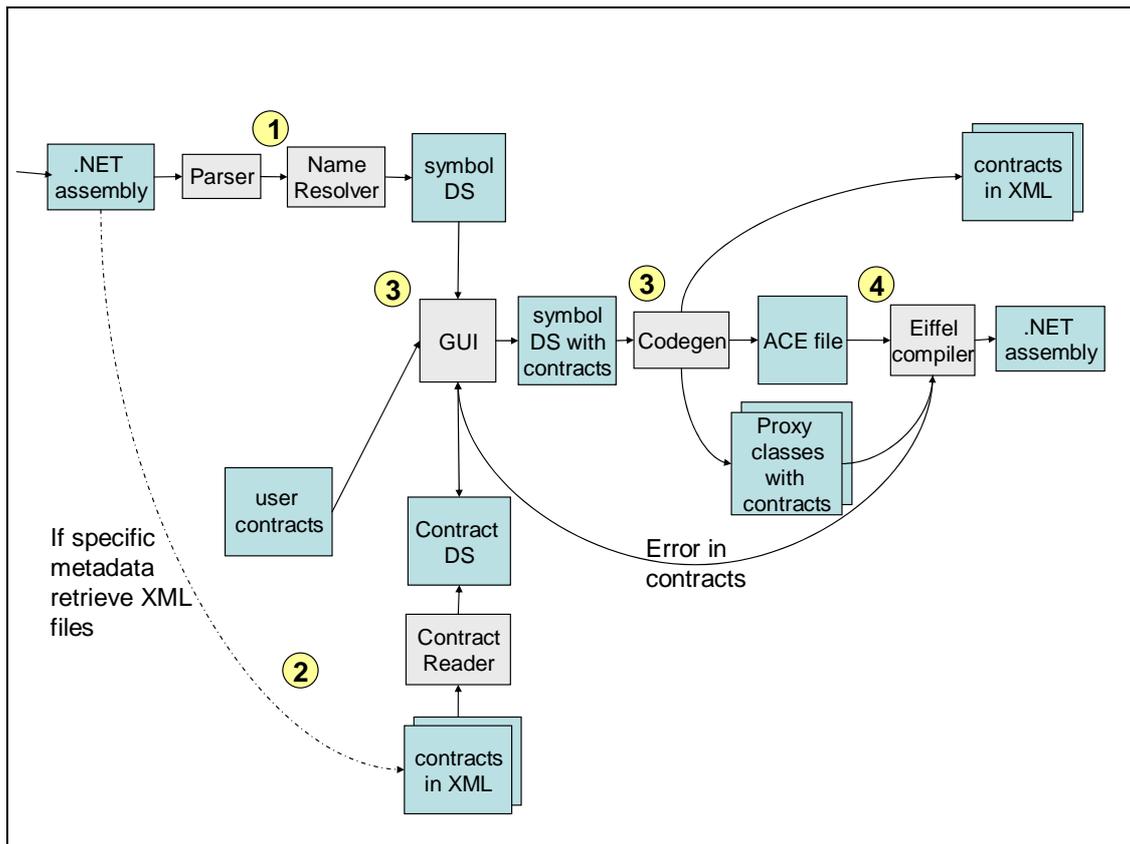
Figure 1: Software architecture

(1) If there is no specific metadata, the parser parses the *.NET* assembly and produces a *symbol DS* (Data Structure). The Name Resolver keeps track of the renaming from *.NET* names to *Eiffel* names.

(2) If there is specific metadata (e.g. a path to the *XML* contract files) in the *.NET* assembly expressing that there are already contracts for it, the Contract Reader reads the contracts from *XML* contract files and produces a Contract DS. Furthermore, like in point (1), a *symbol DS* is produced.

(3) Both the Contract DS and the *symbol DS* are handled by the Contract Handler and the basic *GUI*. They produce a *symbol DS* containing information about the *.NET* assembly and the contracts. The Contract Handler and the basic *GUI* are also responsible for user contracts. User contracts are appended to the Contract DS.

(4) The Code Generator traverses the *symbol DS* with contracts and produces proxy classes with contracts. The proxy classes will contain metadata expressing where the Contract Files are located. Another task of the Code generator is to generate *XML* files storing all information about contracts and to produce the *ACE* files needed by the *Eiffel* compiler.

(5) The *Eiffel* compiler is called by the *GUI* and compiles the generated proxy classes using the *ACE* files. If the contracts are not correct, an error message is displayed and the contracts can be corrected considering the error messages.

1.1.3 Intended results

Table 1 below shows the intended results.

Parser	• Implementation of a parser, which parses a *.NET* assembly and produces a *symbol DS*
Name Resolver	• Implementation of a Name Resolver • Scope: *.NET* and *Eiffel* have different naming conventions; hence, different names for the same type or feature. The Name Resolver keeps track of *.NET* types and *.NET* methods. It uses the *Eiffel* [10] core algorithm for renaming.
Contracts	• Implementation of a *DTD* to store contracts in *XML*
Contract Reader	• Implementation of a Contract Reader based on the mentioned *DTD*
Basic *GUI*	• Implementation of a basic *GUI* for demonstration purposes to enable adding contracts (preconditions, postconditions and invariants)
Contract Handler	• Implementation of the Contract Handler, which is used by the basic *GUI*

Code Generator	• Implementation of a code generator consisting of the following parts:
	   o *XML* Writer: Implementation of an *XML* writer which enables storing contracts in an *XML* file based on the specified *DTD*. (One XML contract file per class).
	   o *ACE* File Producer: Implementation of an *ACE* File Producer, which keeps track of the *.NET* assembly and of the *Eiffel* proxy classes
	   o Proxy Class Generator: Implementation of a Proxy Class Generator, which produces *Eiffel* classes with contracts. Scope: Generation of a proxy class for every class (subclass) in the *.NET* assembly
	   o Assembly Updater: This part updates the proxy classes with metadata (e.g. path to the *XML* contract files)

Demo application	• Implementation of a demo demonstrating the functionality and scope of *Contract Wizard II*
Test cases	• Test cases that will be used for the demo

Table 1: Intended results

1.2 Background material

1.2.1 Reading list

- Chapters in OOSC2 [7] in particular:
    - o 1 Software quality
    - o 11 Design by Contract: building reliable software
    - o 23 Principles of class design
    - o 26 A sense of style
    - o 28 The software construction process

- NET Training Course [5]
- Rapport de stage "Jeune Ingénieur" [1]

1.2.2 Software

- Source code of *Contract Wizard*

1.3 Project management

1.3.1 Objectives and priorities

Table 2 below shows the objectives and priorities of the different parts of the project. One represents the highest priority and three the lowest priority. The most important software parts are: design of the software architecture, the parser, the contract reader, the code generator and the test cases. The most significant documentation part is the thesis report including the developer documentation.

Objective	Priority
Software architecture	1
Parser	1
Contract Reader	1
Contract Handler	1
Code Generator	1
Demo application	2
Basic *GUI*	3
Test cases	1
Optimization	3
User documentation	2
Developer documentation	1
Intermediary report	3
Thesis report	1

Table 2: Objectives and priorities

1.3.2 Criteria for success

The focus on this project is quality. The result may be a partial implementation of the intended results and objectives without implying any penalty on the success of the project.

Quality of software:

- Use of *Design by Contract*
    - Routine pre- and postconditions
    - Class invariants
    - Loop variants and invariants

- Careful design
    - Design Patterns
    - Extendibility
    - Reusability
    - Careful abstraction

- Core principles of OOSC2 [7]
- Command/query separation
    - Simple interfaces
    - Uniform access
    - Information hiding
    - etc.

- Style guidelines
- Correct and robust code
- Readability of source code
- Ease to use

Quality of documentation:
- Completeness
- Understandable documentation
- Usefulness
- Structure

1.3.3 Method of work

The technologies involved are:
- The *System.Reflection* library from the *.NET* Framework [14]
- The *System.XML* library from the *.NET* Framework [14]
- Programming language: *Eiffel for .NET* [11]
- ISE *Eiffel* compiler beta version 5.4 for test purposes [11]
- The development tool is *Eiffel Studio* 5.3 [11]

I developed the software using the *Eiffel* style of design in close cooperation with the supervisor.

1.3.4 Quality management

Quality was ensured by:
- **Weekly progress reports:** Short weekly progress reports sent to the supervisor
- **Milestone progress reports:** Detailed reports for each milestone (see plan with milestones below)
- **Review and Validation:** Review of each milestone by the supervisor concluded by a meeting
- **Validation:** Validation of each milestone after review (see validation steps below)
- **Testing:** Testing of the software by application of different test cases

1.3.5 Documentation

- **Progress reports:**
  - Short weekly progress reports consisting of the main tasks completed
  - Detailed reports for each milestone consisting of:
    - The main tasks
    - Eventual encountered difficulties
    - Implementation, scope of the implementation

- **Developer report**: This manual documents the software architecture and its limitations, describes the difficulties encountered during the implementation, explains how the software could be extended and contains a section discussing the test cases.
- **Intermediary report**: The intermediary report consists of the intermediary developer report.
- **User manual:** The user manual describes the usage of the tool *Contract Wizard II.*
- **Thesis report:** The thesis report consists of the final user manual, the final developer report and a theoretical part discussing the possibility to store the contracts in the assembly metadata.

1.3.6 Validation Steps

The validation for each milestone comprises:
- **Report:** Sending detailed report and the relevant parts of the work to the supervisor for review
- **Meeting:** Organizing a meeting with the supervisor or presentation and discussion of the conducted work
- **Revision:** Revision of parts or all of the work for this milestone, depending on the conclusion of the supervisor

1.4 Plan with milestones

1.4.1 Project steps

Table 3 below shows the milestones of this project.

Milestones	Objectives
M1: Software architecture	Design of the entire Software Architecture (This includes writing the core interfaces of the Parser, the Name Resolver, the Contract Reader, the Contract Handler, the Code Generator, the *symbol DS* and the *symbol DS* with contracts). The Software Architecture is refined in the later milestones.
M2: Parser	Implementation of the Parser, Name Resolver and the *symbol DS*
M3: Contract Reader	Design of the *DTD* and implementation of the Contract Reader based on this *DTD*
M4: Contract Handler	Implementation of the Contract Handler
M5: Code Generator	Implementation of the Code Generator consisting of three parts: • *XML*-Writer • *Ace*-File producer • Proxy Class generator
M6: *Eiffel* Compiler	Implementation of a class calling the *Eiffel* Compiler
M7: Basic *GUI*	Basic *GUI* (only for demonstration purposes)
M8: Demo	Implementation of a demo that illustrates the capabilities of the tool
M9: Intermediary Report	Writing of the intermediary report. Presentation of the intermediary results in a group meeting.
M10: Developer Report	Writing of the developer report
M11: Thesis Report	Writing of the thesis report, including the user documentation, the developer documentation, test case documentation, a theoretical part discussing the possibility to store the contracts in the assembly metadata and a project review. Presentation of the project results in a group meeting.

Table 3: Milestones

1.5 Deadlines

Table 4 below shows the dates of the scheduled deadlines at the beginning of the project and the dates when the milestones have been handled. As you can see, I spent much more time in milestone *M2 Parser* and *M3 Contract Reader* (it turned out to be more complex to parse the *.NET* assemblies and their *XML* representation than initially assumed) which caused a delay for the other milestones. Due to the delay, we decided to move milestone *M4* at the end of the project and continued directly with milestone *M5*. Instead of writing a basic *GUI,* we decided to write a command line interface*.

Milestone	Scheduled Deadline	Handed in
M1 Software architecture	2003-06-13 Fr	2003-06-13 Fr
M2 Parser	2003-07-04 Fr	2003-07-17 Fr
M3 Contract Reader	2003-07-13 Fr	2003-08-06 Fr
M4 Contract Handler	2003-07-25 Fr	2003-10-03 Fr
M5 Code generator	2003-08-08 Fr	2003-09-01 Fr
M6 Eiffel Compiler	2003-08-08 Fr	2003-10-03 Fr
M7 Basic GUI*	2003-08-15 Fr	2003-10-03 Fr
M8 Demo	2003-08-22 Fr	2003-10-03 Fr
M9 Intermediary report	2003-08-29 Fr	2003-10-02 Fr
M10 Developer report	2003-10-06 Fr	2003-10-06 Fr
M11 Thesis report	2003-10-11 Fr	2003-10-06 Fr

Table 4: Deadlines

1.6 Tentative Schedule

Figure 2 on the next page shows the tentative project schedule.

Figure 2: Tentative and real project schedule

| Task	Start	End	2003-06-06 Fr (23)	2003-06-16 Mo (24)	2003-06-23 Mo (25)	2003-06-30 Mo (26)	2003-07-07 Mo (27)	2003-07-14 Mo (28)	2003-07-21 Mo (29)	2003-07-28 Mo (30)	2003-08-04 Mo (31)	2003-08-11 Mo (32)	2003-08-18 Mo (33)	2003-08-25 Mo (34)	2003-09-01 Mo (35)	2003-09-08 Mo (36)	2003-09-15 Mo (37)	2003-09-22 Mo (38)	2003-09-29 Mo (39)	2003-10-06 Mo (40)	2003-10-13 Mo (41)	2003-10-20 Mo (42)
Diploma Project	23	40	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█		
Presentation																						
P1: Introductory presentation	24	24		P1																		
P2: Final presentation	39	41																				P2
Design																						
M1: Software architecture (tentative)				M1																		
M1: Software architecture (real)				M1																		
M3: DTD (tentative)							█															
M3: DTD (real)							█															
Implementation																						
M2: Parser (tentative)					█	M2																
M2: Parser (real)					█	█	M2															
M2: Name resolver (tentative)							█															
M2: Name resolver (real)							█															
M3: Contract Reader (tentative)							M3															
M3: Contract Reader (real)											█	M3										
M4: Contract Handler (tentative)								█	M4													
M4: Contract Handler (real)																		M4				
M5: Code generator (tentative)										M5												
M5: Code generator (real)												█	M5									
M6 Eiffel Compiler (tentative)											M6											
M6 Eiffel Compiler (real)																		M6				
M7: Basic GUI (tentative)												M7										
M7: UI (real)																		M7				
M8: Demo (tentative)													M8						M8			
M8: Demo (real)													M8						M8			
Documentation																						
M9: Intermediary report (tentative)	33	34	█	█	█	█	█	█	█	█	█	█	█	M9								
M9: Intermediary report (real)	33	34																				
M10: Developer report (tentative)	34	35	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	M10				
M10: Developer report (real)	34	35	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	M10			
M11:Thesis report (tentative)	38	42															█	█	M11			
M11:Thesis report (real)	38	42																█	M11			
Testing																						
Test cases (tentative)														█	█	█	█	█				
Test cases (real)														█	█	█	█	█	█	█		
Optimization; optional parts (tentative)														█	█	█	█	█				
Optimization; optional parts (real)																		█				

Figure 2: Tentative and real project schedule

15

Part B: User manual

2. Demo Application

2.1 *Contract Wizard* commands

Figure 3 shows the commands provided by *Contract Wizard II*
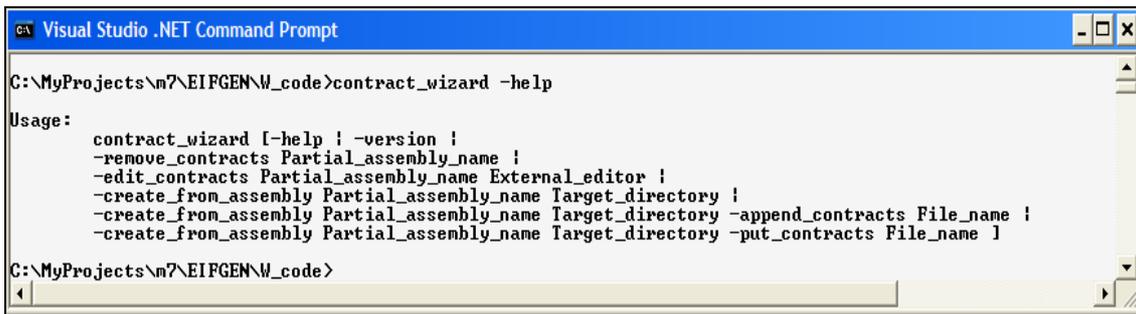
Figure 3: *Contract wizard II* commands

- The command ***help*** shows all valid commands (see *Figure 3*)
- The command ***version*** shows the *Contract Wizard* version
- The command ***remove_contracts*** removes the *XML* file storing all contracts for the assembly specified by ***Partial_assembly_name***
- The command ***edit_contracts*** opens the file containing the *XML* representation of the *Contract Wizard* proxy classes in an external editor. If you use for example the editor *vi* you can edit the contracts using the command ***edit_contracts my_assembly vi***
- The command ***create_from_assembly*** means that we want to create *Contract Wizard* proxy classes for the assembly specified by ***Partial_assembly_name***
- The command ***append_contracts*** inserts contracts defined in file ***File_name*** into the assembly corresponding to ***Partial_assembly_name***.
- The command ***put_contracts*** removes all existing contracts and adds new contracts specified in the file (***File_name***).

Partial_assembly_name describes only a part of the assembly name. The "assembly fully qualified name" also contains the assembly version and the assembly public key token. In the current version, *Contract Wizard* loads the newest version of the assembly from the global assembly cache (*GAC*) specified by the ***Partial_assembly_name***.

2.2 *Contract Wizard* accounting example

The *Contract Wizard* provides users with the possibility to add contracts to a *CLS* compliant *.NET* assembly a posteriori. The tool can be used incrementally, meaning it is possible to add contracts to a *.NET* assembly that has already been contracted using the *Contract Wizard*. We

17

will explain how to add contracts to a *.NET* assembly on the *accounting* example delivered with this project. In the directory "*./example/Accouting_Example/ C#_sources_for_Account.dll/*" you find the files *Account.dll, c1.csv* (the extension "*.csv*" stands for comma separated values) and *c2.csv* (the latter two contain the contracts).

Table 5 below shows the class *Account* from assembly *Account.dll* to which we want to add contracts.

```csharp
using System;

[assembly: CLSCompliant ( true )]

namespace Accounting
{
    public class Account
    {
        // Set up account with `InitialAmount'.
        public Account( int InitialAmount )
        {
            Balance = InitialAmount;
            Deposits = new DepositList();
            Withdrawals = new WithdrawalList();
        }

        // Deposit `Sum' into the account.
        public void Deposit( int Sum )
        {
            Add( Sum );
            Deposits.Add( new Deposit( Sum ) );
        }

        // Withdraw `Sum' from the account.
        public void Withdraw( int Sum )
        {
            Add( -Sum );
            Withdrawals.Add( new Withdrawal( Sum ) );
        }

        // Is it possible to withdraw `Sum' from the account?
        public bool MayWithdraw( int Sum )
        {
            return ( Balance >= MinimumBalance + Sum );
        }

        // Add `Sum' to `Balance'.
        private void Add( int Sum )
        {
            Balance = Balance + Sum;
        }
```

```
                    // Account balance
                    public int Balance;

                    // Minimum balance
                    static public int MinimumBalance = 1000;

                    // List of deposits
                    public DepositList Deposits;

                    // List of withdrawals
                    public WithdrawalList Withdrawals;
                }
            }
```

Table 5: C# class *Account.cs*

In this example, we add the precondition (in Eiffel syntax), *not_too_big*: *a_sum <= balance - minimum_balance* to feature *withdraw* of class *Account*. The file *c1.csv* contains the contract *not_too_big* (see *Figure 4* below).

2.2.1 Insertion of a contract

We add the contract *not_too_big* using the command line application *contract_*wizard (see *Figure 4* below). The first command *-create_from_assembly Account c:\cw_tmp* specifies from which assembly we want to create the *Contract Wizard* proxy classes and where they have to be stored. The second command *–append_contracts c1.csv* specifies that we would like to append new contracts stored in the file *c1.csv*.

Figure 4: Insertion of a contract with *Contract Wizard II*

Table 6 shows the generated *Contract Wizard* proxy class (*CW_ACCOUNT*) for the class *Account*. A closer look at *Table 6* on the next page shows the inserted precondition (1).

```
    indexing

            note: "Automatically generated by the Contract Wizard."
            dotnet_name: "Accounting.Account"

    class CW_ACCOUNT
```

```eiffel
    inherit

        SYSTEM_OBJECT
                redefine
                        get_hash_code,
                        equals,
                        to_string
                end

create

        make

feature {NONE} -- Initialization

        frozen make (a_initial_amount: INTEGER) is
                        -- dotnet_name: "Account..ctor (InitialAmount: Int32)"
                do
                        create account_ref.make (a_initial_amount)
                end

feature -- Access

        frozen balance: INTEGER is
                        -- dotnet_name: "Account.Balance: Int32"
                do
                        Result := account_ref.balance
                end

        frozen deposits: DEPOSIT_LIST is
                        -- dotnet_name: "Account.Deposits: DepositList"
                do
                        Result := account_ref.deposits
                end

        frozen withdrawals: WITHDRAWAL_LIST is
                        -- dotnet_name: "Account.Withdrawals: WithdrawalList"
                do
                        Result := account_ref.withdrawals
                end

        frozen minimum_balance: INTEGER is
                        -- dotnet_name: "Account.MinimumBalance: Int32"
                do
                        Result := account_ref.minimum_balance
                end
```

feature -- Query

    frozen *get_hash_code*: *INTEGER* **is**
        -- dotnet_name: "Account.GetHashCode (): Int32"
      do
        Result := *account_ref.get_hash_code*
      end

    frozen *equals* (*a_obj*: *SYSTEM_OBJECT*): *BOOLEAN* **is**
        -- dotnet_name: "Account.Equals (obj: Object): Boolean"
      do
        Result := *account_ref.equals* (*a_obj*)
      end

    frozen *to_string*: *SYSTEM_STRING* **is**
        -- dotnet_name: "Account.ToString (): String"
      do
        Result := *account_ref.to_string*
      end

    frozen *may_withdraw* (*a_sum*: *INTEGER*): *BOOLEAN* **is**
        -- dotnet_name: "Account.MayWithdraw (Sum: Int32):
        -- Boolean"
      do
        Result := *account_ref.may_withdraw* (*a_sum*)
      end

    frozen *get_type_from_original_class*: *TYPE* **is**
        -- dotnet_name: "Account.GetType (): Type"
      do
        Result := *account_ref.get_type*
      end

feature -- Commands

    frozen *deposit* (*a_sum*: *INTEGER*) **is**
        -- dotnet_name: "Account.Deposit (Sum: Int32)"
      do
        account_ref.deposit (*a_sum*)
      end

    frozen *withdraw* (*a_sum*: *INTEGER*) **is**
        -- dotnet_name: "Account.Withdraw (Sum: Int32)"
      require
        not_too_big: *a_sum <= balance - minimum_balance* (1)
      do
        account_ref.withdraw (*a_sum*)
      end

```
        feature {NONE} -- Implementation

             frozen account_ref: ACCOUNT
                        -- Reference to the .NET class

        end
```

Table 6: Proxy class *CW_ACCOUNT* with a precondition on feature *withdraw*

2.2.2 Incrementality

To demonstrate incrementality we insert a second contract (see *Figure 5* below). The invariant *not_under_minimum*: *balance >= minimum_balance* states that the balance is always bigger than the minimum balance.

Figure 5: Adding a new contract

Table 6 shows a fragment of the modified proxy class *CW_ACCOUNT*. The class contains the inserted contracts *not_too_big* (1) and *not_under_minimum_balance* (2).

```
        Indexing

             note: "Automatically generated by the Contract Wizard."
             dotnet_name: "Accounting.Account"

        class CW_ACCOUNT

        inherit

             SYSTEM_OBJECT
                  redefine
                       get_hash_code,
                       equals,
                       to_string
                  end

        create

             make
```

22

```
    feature {NONE} -- Initialization

        frozen make (a_initial_amount: INTEGER) is
                -- dotnet_name: "Account..ctor (InitialAmount: Int32)"
            do
                create account_ref.make (a_initial_amount)
            end

…

    feature -- Commands

        frozen deposit (a_sum: INTEGER) is
                -- dotnet_name: "Account.Deposit (Sum: Int32)"
            do
                account_ref.deposit (a_sum)
            end

        frozen withdraw (a_sum: INTEGER) is
                -- dotnet_name: "Account.Withdraw (Sum: Int32)"
            require
                not_too_big: a_sum <= balance - minimum_balance (1)
            do
                account_ref.withdraw (a_sum)
            end

    feature {NONE} -- Implementation

        frozen account_ref: ACCOUNT
                -- Reference to the .NET class

    invariant

        not_under_minimum: balance >= minimum_balance (2)

end
```

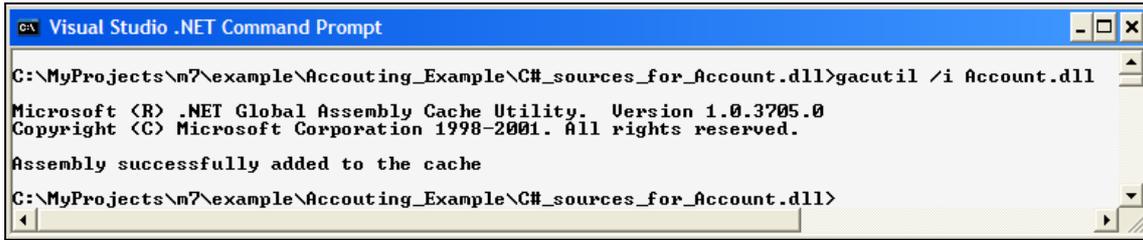**Table 7: Proxy class *CW_ACCOUNT* with an invariant**

2.2.3 **Adding assemblies to the *Global Assembly Cache***

Contract Wizard reads assemblies from the Global Assembly Cache. To add an assembly into the *GAC* use the *gacutil* tool delivered with the *.NET* Framework (see Figure 6 below).

Figure 6: Adding an assembly into the *Global Assembly Cache*

Part C: Developer manual

3. Software architecture

In this chapter, I give a rough overview of the software architecture of the Contract Wizard. First, I describe briefly the cluster hierarchy of the system. Second, I focus on the *AST* sub cluster and explain it in more details.

3.1 Cluster overview

3.1.1 The *root_cluster*

Contract Wizard II consists of the following clusters:
- *controller*: classes for handling the *symbol DS* and its contracts
- *doc*: *Contract Wizard* documentation, including this report
- *example*: some example applications, including the *Accounting* example presented before
- *generation*: classes for code generation (see also section *The cluster generation*)
- *parse*: classes for parser generation - *.NET* parser and *XML* parser (see also section *The sub cluster parse/ast.   Ast* stands for Abstract syntax tree.)

3.1.2 Sub clusters of the cluster *parse*

The cluster *parse* contains the following sub clusters:
- *ast*: classes representing the *symbol DS*
- *factory*: classes to produce the *symbol DS*
- *formatter*: classes responsible for translating *.NET* names into *Eiffel* names
- *incrementality*: classes indicating whether there are contracts for the specified *.NET* assembly or not (see also *Sub cluster incrementality: dealing with already contracted .NET assemblies*)
- *parser*: classes responsible for parsing *.NET* assemblies and their corresponding *XML* files
- *support*: helper classes providing functionalities used by other classes in other clusters.
- *visitor*: classes implementing the *visitor pattern* [3]

3.1.3 The cluster *generation*

The cluster *generation* has two sub clusters: *internal* and *dotnet*. The sub cluster *internal* contains the following classes:

- ***CW_LACE_GENERATOR***
  This class generates an *ACE* file from the *symbol DS*.

- ***CW_EIFFEL_GENERATOR***
  This class generates the proxy classes from the *symbol DS*.

- ***CW_XML_GENERATOR***
  This class generates an *XML* file representing the *symbol DS*.

- ***CW_GENERATOR***
  The above classes have in common that they all need a *symbol DS* and a specific *visitor*, which they inherit from the class *CW_GENERATOR*. Code is produced by traversing the *symbol DS*.

  To simplify code generation I use the *visitor pattern* [3]. The class *CW_EIFFEL_VISITOR* helps to produce *Eiffel* code from the *symbol DS*. The class *CW_XML_VISITOR* helps to generate the *XML* representation of the *symbol DS*. Both classes will be discussed thoroughly in the section *Accessing the symbol DS*.

The sub cluster *dotnet* includes the following classes:

- ***CW_COMPILER_LAUNCHER***
  This class launches the compiler and reports the user interface with possible compilation error messages.

- ***CW_EDITOR_LAUNCHER***
  This class launches an external editor.

3.2 The sub cluster *parse/ast*

3.2.1 Contract Wizard type (*CW_TYPE*)

The *symbol DS* contains the data to produce the *Eiffel* proxy classes and their *XML* representation. The *symbol DS* is a list of elements of type *CW_TYPE*. An instance of *CW_TYPE* has features (*CW_FEATURE*), interfaces (*CW_INTERFACE*) and invariants (*CW_INVARIANT*), where a feature can either be a routine (*CW_ROUTINE*) or an attribute (*CW_ATTRIBUTE*) (see *Figure 7* below).
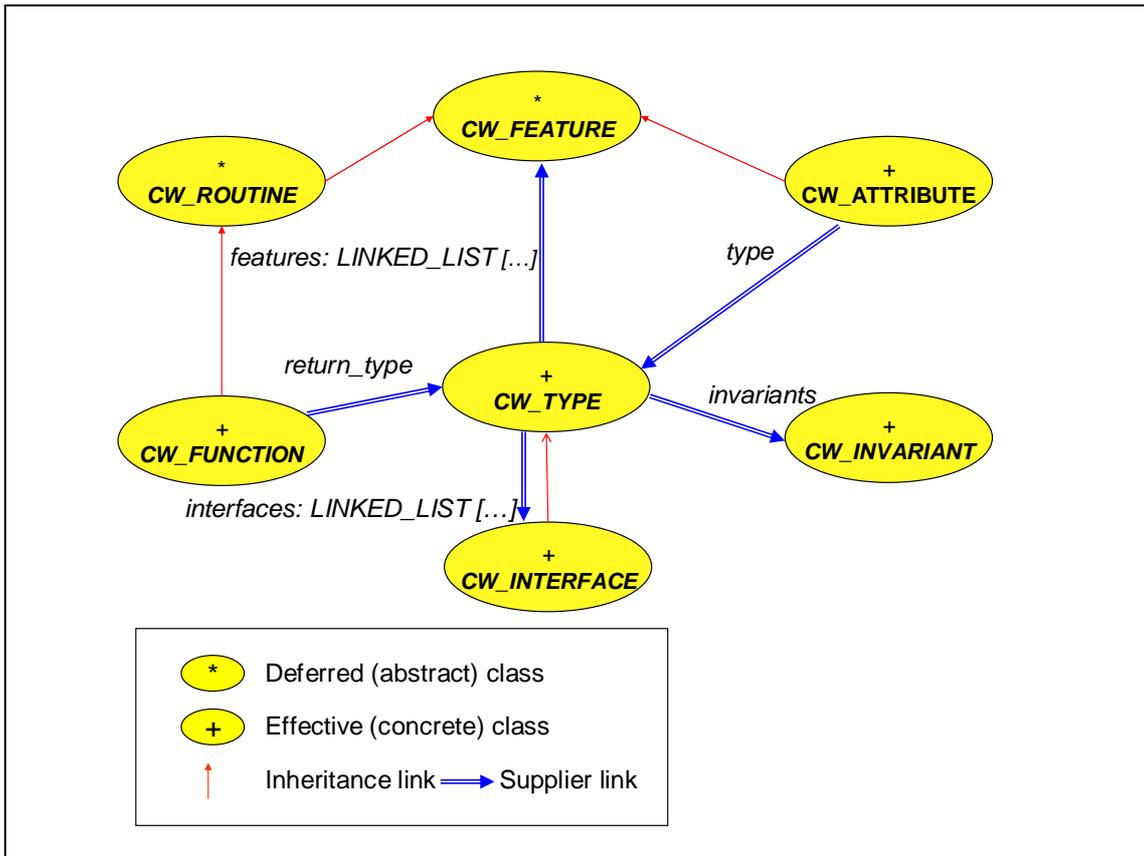
Figure 7: *CW_TYPE* **and its features, interfaces and invariants**

3.2.2 Contract Wizard Interface (*CW_INTERFACE*)

As shown on *Figure 7 CW_INTERFACE* inherits from *CW_TYPE*. *CW_INTERFACE* has an additional access feature *special_names* (*LINKED_LIST* [*CW_NAME*]). This feature returns the interface feature names, which has to be undefined in the proxy class later (see also Chapter 7). The reason I use *CW_INTERFACE* instead of *CW_TYPE* is the feature *special_names* that facilitates the creation and the parsing of the *XML* representation of the proxy classes. *Special_names* is a linked list storing the interface feature names, which are foreseen to be undefined in the proxy class

3.2.3 Contract Wizard feature (*CW_FEATURE*)

The next version of *Eiffel* [11] will allow assertions on attributes (*CW_ATTRIBUTE*). That is why a feature (*CW_FEATURE*) has preconditions (*CW_PRECONDITION*) and postconditions (*CW_POSTCONDITION*). A feature is either a routine (*CW_ROUTINE*) or an attribute (*CW_ATTRIBUTE*) (see *Figure 8* below).
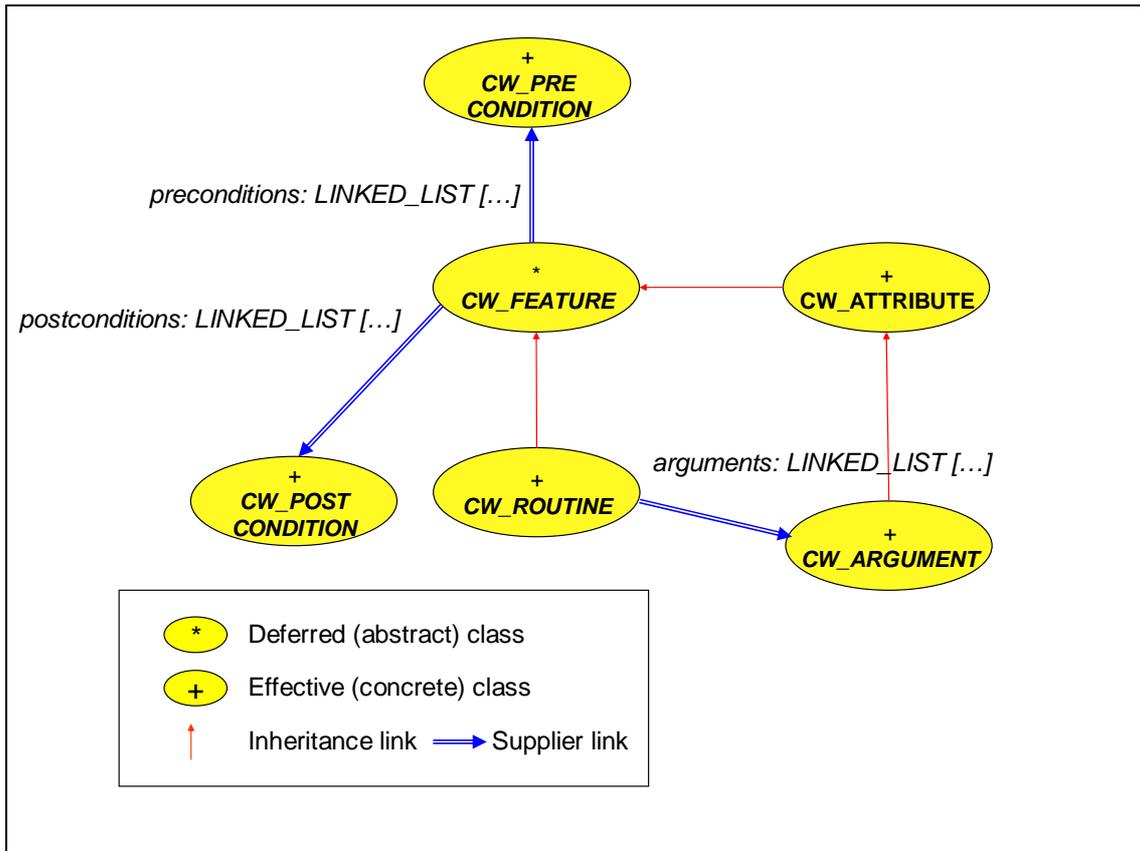
Figure 8: Clients of *CW_FEATURE*

3.2.4 Contract Wizard routine (*CW_ROUTINE*)

A routine has a list of arguments (*CW_ARGUMENT*) (see *Figure 8*). A routine is either a procedure (*CW_PROCEDURE*) or a function (*CW_FUNCTION*) (see *Figure 9* on next page); the latter has a return type (*CW_TYPE*).

3.2.5 Contract Wizard attribute (*CW_ATTRIBUTE*)

Every attribute has a name (*CW_NAME*) and a type (*CW_TYPE*).

3.2.6 Implementation hierarchy

CW_TYPE, CW_FEATURE and *CW_ASSERTION* inherit the feature *visit* from *CW_AST_NODE*. We need this feature to implement the *visitor pattern* [3].

The class *CW_FEATURE* introduces preconditions and postconditions, which are inherited by *CW_ROUTINE* and *CW_ATTRIBUTE*.

Both *CW_FUNCTION* and *CW_PROCEDURE* inherit from *CW_ROUTINE*. A creation procedure is a special procedure. That is why *CW_CREATION_PROCEDURE* inherits from *CW_PROCEDURE*.

28

All preconditions (*CW_PRECONDITION*), postconditions (*CW_POSTCONDITION*) and invariants (*CW_INVARIANT*) have an assertion tag and an assertion expression in common. They inherit the assertion tag and the assertion expression from *CW_ASSERTION*.
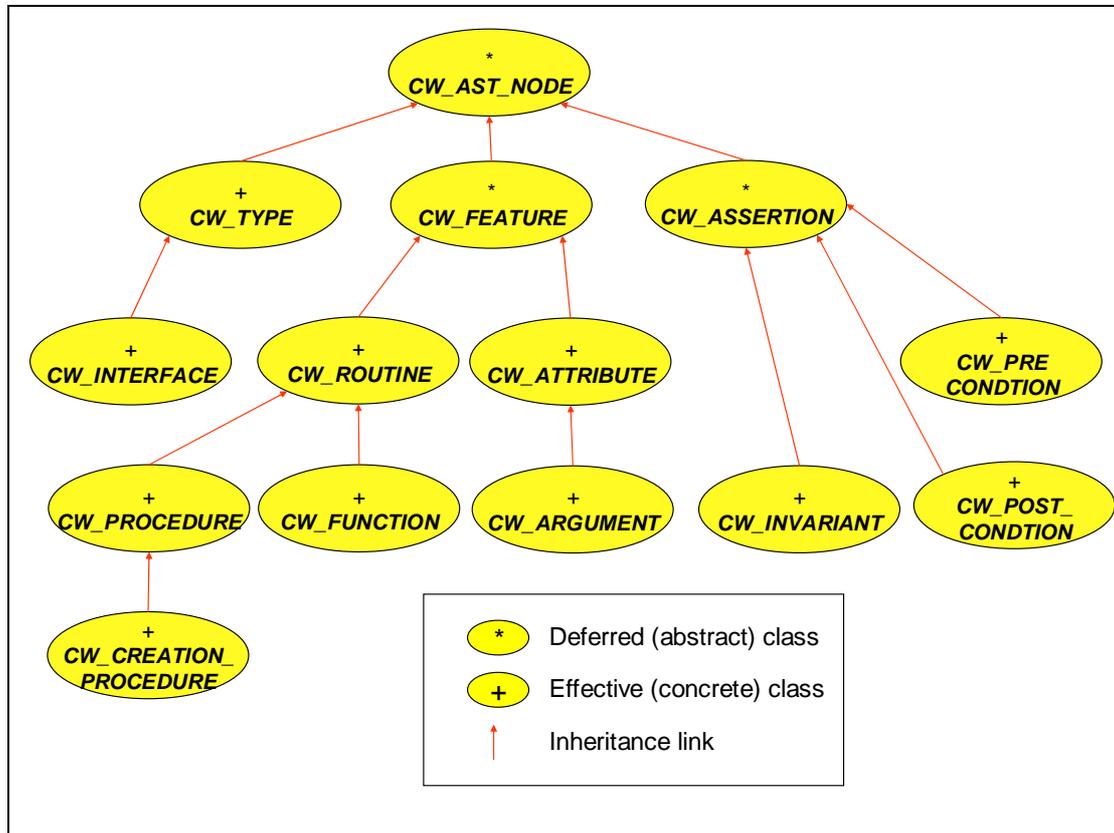
Figure 9: *AST* **used by the** *Contract Wizard*

3.2.7 The sub cluster *parse/factory*: producing the *symbol DS*

I introduce a class *CW_FACTORY* (with its two variants *CW_AST_FACTORY* and *CW_DOTNET_FACTORY*) to make the creation of *AST* nodes simpler. (This corresponds to the Abstract Factory pattern described in [3]). *Ast* nodes can only be created by the *CW_AST_FACTORY*.

3.2.8 The sub cluster *parse/formatter*: translating .*NET* names into *Eiffel* names

This cluster contains the class *CW_NAME_FORMATTER*, which translates .*NET* names into *Eiffel* names.

3.2.9 Sub cluster *incrementality*: dealing with already contracted *.NET* assemblies

If the user has added contracts to a *.NET* assembly, there is an *XML* file containing all information about the generated proxy classes and their contracts. In this case, the *Contract Wizard* gets the *symbol DS* from the *XML* file. (Otherwise it retrieves the *symbol DS* from the .NET assembly). The cluster *incrementality* contains all classes used to determine whether an assembly already has some contracts.

3.2.10 The sub cluster *parse/parser*

This cluster contains the effected classes *CW_XML_PARSER* and *CW_DOTNET_PARSER*. Both inherit from *CW_PARSER*. The role of the class *CW_XML_PARSER* is to parse an *XML* document and to produce the *symbol DS*. The role of the class *CW_DOTNET_PARSER* is to produce the *symbol DS* from a *.NET* assembly using the reflection mechanism of *.NET*. In both cases we end up with a *symbol DS*. The only difference is that in the first case (*CW_XML_PARSER*) the *symbol DS* contains contracts.

3.2.11 The sub cluster *parse/support*

This cluster contains classes that provide facilities needed by other classes of the same 'parse' cluster. It contains in particular constant classes.

3.2.12 The sub cluster *visitor*: accessing the *symbol DS*

There are two possibilities to access data stored in the *symbol DS*. Either we access the data directly using the desired features (e.g. *a_type.features.name*) or we use a *visitor*. The table below shows how to use a *visitor*. In this example a type and its features are visited. Then, there are two assignments. The first assignment retrieves a list of attributes; the second assignment retrieves a list of procedures. Both lists have been extended during the loop.

```
type.visit (visitor)
features := type.features
from
        features.start
until
        features.after
loop
        features.item.visit (visitor)
        features.forth
end
attributes := visitor.attributes
procedures := visitor.procedures
```

Table 8: Example of how to use a visitor

Table 9 shows an example of how the lists could be extended. For every call *a_symbol.visit* (*visitor*) a specific visitor procedure is called on the *visitor* object. If the visited symbol in the *symbol DS* is of type *CW_PROCEDURE* then the visitor procedure *visit_procedure* is called, if the visited symbol is of type *CW_ATTRIBUTE* then the visitor procedure *visit_attribute* is called, and so on.

Back to the example this means that we can create specific *Eiffel* code or its *XML* representation for every visited symbol through execution of the specific visitor procedure. From the argument *symbol* of the visitor procedure we can retrieve all needed information to produce code (for example if the symbol is of type *CW_PROCEDURE* we can get the *procedure name*, the *attributes* …).

```
class EIFFEL_VISITOR_EXAMPLE

inherit

        VISITOR

create

        make

feature -- Basic operations

        visit_procedure (a_procedure: CW_PROCEDURE) is
                        -- Visit procedure a_procedure.
                local
                        a_procedure_name: STRING
                        some_attributes: STRING
                do
                        …
                        a_procedure_name := a_procedure.eiffel_name
                        …
                        a_procedure_name.append (some_attributes)
                        procedures.extend (a_procedure_name)
                        …
                end

        visit_creation_procedure (a_creation_procedure: CW_PROCEDURE) is
                        -- Visit creation procedure a_creation_procedure.

        …

        end
```

Table 9: Example using the *EIFFEL_VISITOR* class

4. The .NET parser

This chapter starts by describing some parts of the *.NET* Framework that are directly relevant to *Contract Wizard II*. Then, it explains the implementation of the *.NET* parser that parses *.NET* assemblies using the *.NET* reflection capabilities.

4.1 The *.NET* Framework

4.1.1 *.NET* Framework overview

The *.NET* Framework
Figure 10 shows the architecture of the *.NET* Framework. The two main components in the *.NET* Framework are the Common Language Runtime (*CLR*) and the *.NET* Framework class library. *The CLR is a modern runtime environment that manages the execution of user code* [16]. *The .NET Framework class library is a library of classes providing access to system functionality and is designed to be the foundation on which .NET Framework applications, components and controls are built* [14].

The Common Type System (*CTS*)
The Common Language Specification (*CLS*) and the Common Type System (*CTS*) are based on the *.NET* Framework class library.
The Common Type System (CTS) is a shared type system. The CTS defines the rules by which all types are declared, defined and managed, regardless of source language [16].

The Common Language Specification (CLS)
The Common Language Specification (*CLS*) is a subset of the *CTS* (see *Figure 11*) that enables cross language integration. The *.NET* languages, among them *Eiffel for .NET*, Visual Basic.*NET* and C#, are based on the *CLR*/*CTS*.
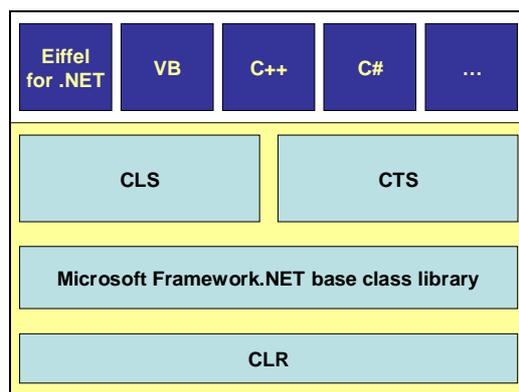
Figure 10: Architecture of the *.NET* Framework

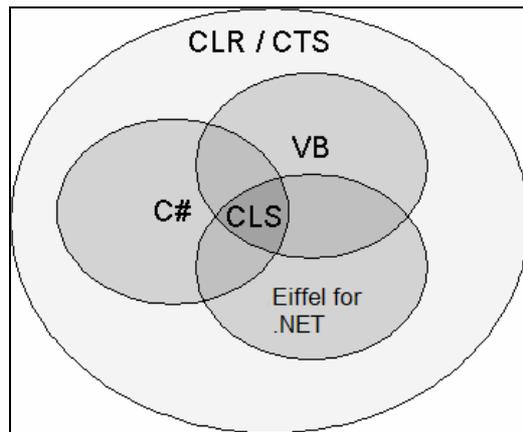### 4.1.2 The Common Language Specification (CLS)

Figure 11: Relations between the CLR/CTS and the CLS

Ideal would be cross language integration based on the *CTS*, but the *CTS* is too large for most non OO-languages. That's why the *CLS* - a subset of the *CTS* (see *Figure 11*) - has been introduced. The *CLS* is part of the *ECMA* standard [10]. Every language can be a *CLS* consumer, a *CLS* extender or a *CLS* producer.

- A language is a **CLS consumer** if it can consume any *CLS* compliant type.
- A language is a **CLS extender** if it can extend any *CLS* compliant type.
- A language is a **CLS producer** if it produces *CLS* compliant code.

Eiffel for .NET is a *CLS* consumer, a *CLS* extender and a *CLS* producer.

4.1.3   Contract Wizard II and the .NET Framework

Contract Wizard II is based on cross-language interoperability: The generated proxy classes are in *Eiffel for .NET*. The resulting *.NET* assembly (obtained by compiling the generated *Eiffel for .NET* proxy classes) can be used from any *.NET* language.

Since only *CLS* compliant code guarantees cross language interoperability among all *.NET* languages, it is required that the original *.NET* assembly and the proxy classes are *CLS* compliant.

4.2   *.NET* parser class overview

To parse a *.NET* assembly I implemented the following classes:

- *CW_PARSER*: A class providing common features for the *.NET* parser (*CW_DOTNET_PARSER*) and the *XML* parser (*CW_XML_PARSER*).
- *CW_DOTNET_PARSER*: A class implementing the *.NET* parser, which populates the *symbol DS* by retrieving information from the *.NET* assembly given as input using the reflection capabilities of *.NET*.
- *CW_XML_PARSER*: A class implementing the *XML* .parser, which reads an *XML* representation of the *symbol DS* - that may contain contracts - and builds the corresponding *AST* to be used by the Contract Handler.

- *CW_AST_FACTORY*: A class providing common features for the class *CW_DOTNET_AST_FACTORY* and the class *CW_XML_AST_FACTORY*.
- *CW_DOTNET_AST_FACTORY*: A class responsible for creating *AST* nodes (*CW_TYPE*, *CW_PROCEDURE*, *CW_ATTRIBUTE*, …) from *.NET* members (*METHOD_INFO, FIELD_INFO*, ...) and *.NET* types (*TYPE*).
- *CW_NAME_FORMATTER*: A class responsible for translating *.NET* member names and *.NET* type names into *Eiffel* feature names and *Eiffel* type names.
- *CW_DOTNET_KEYWORD_CONSTANTS*: A class defining *.NET* keywords.
- *CW_DOTNET_TYPE_CONSTANTS*: A class defining *.NET* type constants.
- *CW_EIFFEL_KEYWORD_CONSTANTS*: A class defining *Eiffel* keywords.
- *CW_PUNCTUATION_CONSTANTS*: A class defining punctuation constants.

4.3 The class *CW_AST_FACTORY*

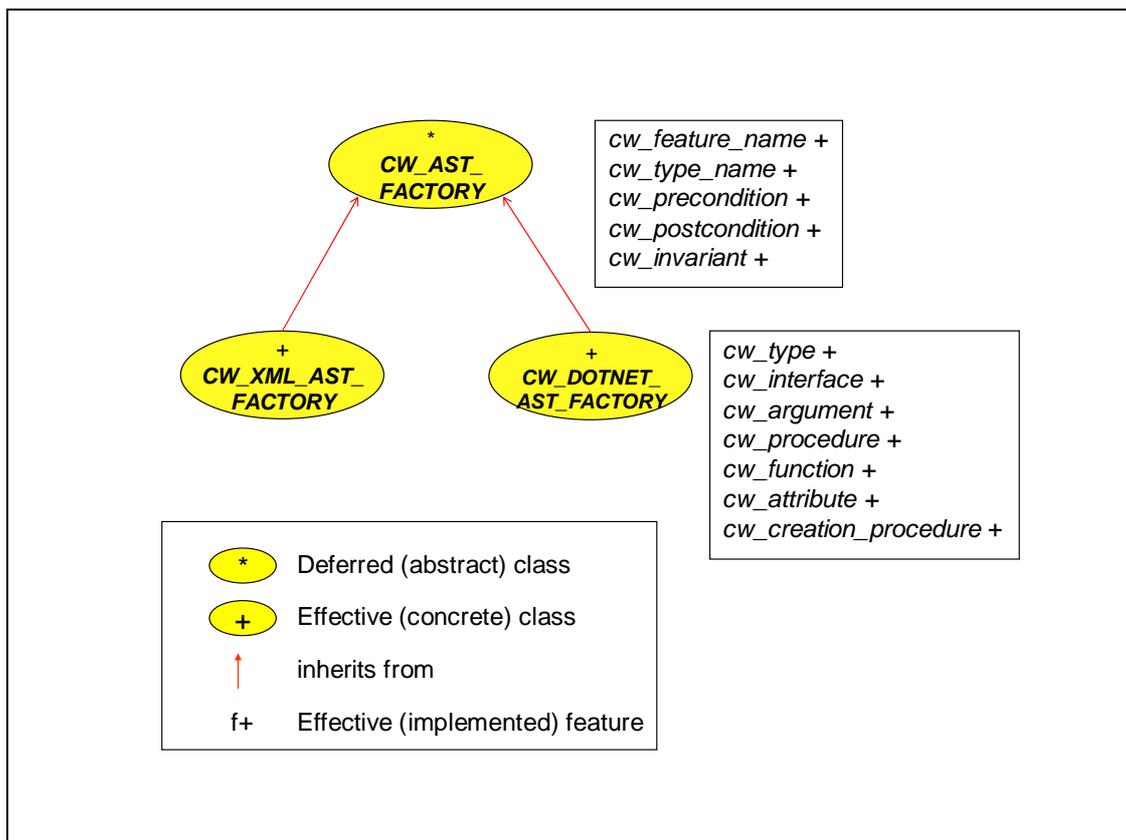### 4.3.1 Generating *AST* nodes for contracts, feature and type names

Figure 12: *Contract Wizard AST* factory classes

I have implemented a *factory* (*factory pattern* [3]) to facilitate the creation of *Contract Wizard AST* nodes (*CW_PROCEDURE, CW_FUNCTION, CW_TYPE...*). The *factory* consists of the following classes: *CW_AST_FACTORY, CW_XML_AST_FACTORY* and *CW_DOTNET_AST_FACTORY* (see *Figure 12*).

The class *CW_AST_FACTORY* implements the common features for *CW_XML_AST_FACTORY* and CW_*DOTNET_AST_FACTORY*.

The role of class *CW_DOTNET_AST_FACTORY* is to create *Contract Wizard AST* nodes from *.NET* members (*METHOD_INFO, FIELD_INFO...*) and *.NET* types (class *TYPE* corresponding to the *.NET* type *System.Type*).

The role of class *CW_XML_AST_FACTORY* is to create *Contract Wizard AST* nodes (*CW_PROCEDURE, CW_FUNCTION, CW_TYPE...*) from *XML* nodes (*XML_XML_NODE* corresponding to *System.Xml.XmlNode*). Every *XML* node represents a *.NET* member, a *.NET* type or a contract (precondition, postcondition or invariant).

The class *CW_AST_FACTORY* implements the following features: *cw_feature_name, cw_type_name*, *cw_precondition, cw_postcondition, cw_invariant* (see *Figure 12* above). They are described below.

4.3.2 The feature *cw_precondition*

```
cw_precondition (a_tag: like tag; an_expression: like expression):
        CW_PRECONDITION is
            -- Precondition with tag a_tag
            -- and boolean expression an_expression
    require
            tag_not_void: a_tag /= Void
            tag_not_empty: not a_tag.is_empty
            expression_not_void: an_expression /= Void
            expression_not_empty: not an_expression.is_empty
    do
            create Result.make (a_tag, an_expression) (1)
    ensure
            precondition_not_void: Result /= Void
            tag_set: Result.tag = a_tag
            expression_set: Result.expression = an_expression
    end
```

Table 10: Feature *cw_precondition*: creating a precondition from given tag and expression

The feature *cw_precondition* creates a *Contract Wizard* precondition from a contract tag and a contract expression (1).

4.3.3 The feature *cw_postcondition*

The feature *cw_postcondition* creates a *Contract Wizard* postcondition from a contract tag and a contract expression.
The implementation resembles the implementation of *cw_precondition* given above, simply replace "precondition" with "postcondition".

36

4.3.4 The feature *cw_invariant*

The feature *cw_invariant* creates a *Contract Wizard* invariant from a contract tag and a contract expression. The implementation resembles the implementation of *cw_precondition* shown before, just replace "precondition" with "invariant".

4.3.5 The features *cw_type_*name and *cw_feature_name*

```
cw_feature_name (a_name: SYSTEM_STRING): CW_NAME is (1)
        -- Feature name (with Eiffel name and .NET name) created from
        -- a_name
    require
        a_name_not_void: a_name /= Void
    do
        …
        create Result.make (eiffel_name, dotnet_name)
    ensure
        result_not_void: Result /= Void
    end

cw_type_name (a_name: SYSTEM_STRING): CW_NAME is (2)
        -- Type name (with Eiffel name and .NET name) created from
        -- a_name
    require
        a_name_not_void: a_name /= Void
    do
        …
        create Result.make (eiffel_name, dotnet_name)
    ensure
        result_not_void: Result /= Void
    end
```

Table 11: Features *cw_feature_name* and *cw_type_name*: Generating a feature name and a type name

Features *cw_type_*name and *cw_feature_name* create a *Contract Wizard* name (*CW_NAME*) consisting of an *Eiffel* name and a *.NET* name. The reason why there are two features (1, 2) for creating *Contract Wizard* names is performance: *.NET* types and *.NET* members are translated differently into *Eiffel*.

4.4 The class *CW_DOTNET_AST_FACTORY*

4.4.1 Generating *AST* nodes for features and types

The class *CW_DOTNET_AST_FACTORY* implements the following core features: *cw_type, cw_interface, cw_argument, cw_procedure, cw_function, cw_attribute* and *cw_creation_procedur*e (see *Figure 12* above). They are described below.

4.4.2 The feature *cw_type*

```
cw_type (a_type: TYPE): CW_TYPE is (1)
                -- Contract Wizard type (AST node) created from .NET
                -- type a_type
        require
                a_type_not_void: a_type /= Void
                type_name_not_void: a_type.name /= Void
        local
                a_type_name: CW_NAME
                …
        do
                ...
                a_type_name := cw_type_name (a_type.name.to_string)
                create Result.make (a_type_name) (2)
                Result.set_deferred (a_type.is_abstract or a_type.is_interface) (3)
                Result.set_expanded (a_type.is_value_type)
                Result.set_array (a_type.is_array)
                Result.set_enum (a_type.is_enum)
                Result.set_namespace (a_namespace) (4)
                …
                        -- Add interfaces this type implements. (5)
                …
        ensure (6)
                result_not_void: Result /= Void
                deferred_set: Result.is_deferred = a_type.is_abstract or
                        a_type.is_interface
                array_set: Result.is_array = a_type.is_array
                enum_set: Result.is_enum = a_type.is_enum
                expanded_set: Result.is_expanded = a_type.is_value_type
        end
```

Table 12: Feature *cw_type*: Generating a *Contract Wizard* type

The feature *cw_type* creates a *Contract Wizard t*ype (*CW_TYPE*) from a *.NET* type (*TYPE*) (1). The creation of a *Contract Wizard* type requires a *Contract Wizard* type name (*CW_NAME*); therefore, it creates a *Contract Wizard* type name and generates a *Contract Wizard* type with it (2).

To create proxy classes we have to distinguish between the following types: A type can be an array, an enumeration type, an expanded type or a deferred type. The feature *cw_type* sets the following features for the created *Contract Wizard* type depending on the parsed *.NET* type: *is_deferred, is_expanded, is_array* and *is_enum* (3). The feature *is_deferred* is set to true if the parsed type is abstract, *is_expanded* is set to true if the parsed type is a value type, *is_array* is set to true if the parsed type is an array and *is_enum* is set to true if the parsed type is an enumerated type. The namespace is set because it is required for the class description of the proxy classes later (4). We also have to set all *CLS* compliant interfaces the type implements. (5)

The postcondition of *cw_type* ensures that all *Contract Wizard* type features are set (6).

38

4.4.3 The feature *cw_interface*

```
cw_interface (an_interface: TYPE): CW_INTERFACE is
        -- Contract Wizard type (AST node) created from .NET
        -- argument an_argument
    require
        interface_not_void: an_interface /= Void
    local
        …
        an_interface_name: CW_NAME
        a_feature_name: CW_NAME
        …
    do
        …
        create Result.make (an_interface_name) (1)
        …
        Result.add_speial_name (a_feature_name)
    ensure
        result_not_void: Result /= Void
    end
```

Table 13: Feature *cw_interface*: Generating a *Contract Wizard* interface

The feature *cw_interface* creates a *Contract Wizard* interface with a *Contract Wizard* interface name (*CW_NAME*) (1) and sets special interface feature names (*ToString, Equals, GetHashCode*) required by the *Contract Wizard* proxy class generator (2).

4.4.4 The feature *cw_argument*

```
cw_argument (an_argument: PARAMETER_INFO): CW_ARGUMENT is
        -- Contract Wizard type (AST node) created from .NET
        -- argument an_argument
    require
        argument_not_void: an_argument /= Void
        argument_name_not_void: an_argument.name /= Void
    local
        an_argument_name: CW_NAME
        a_type: TYPE
    do
        …
        an_argument_name := cw_feature_name
                (an_argument.name.to_string)
        create Result.make (an_argument_name) (1)
        a_type := an_argument.parameter_type
        Result.set_type (cw_type (a_type)) (2)
```

```
            ensure
                    result_not_void: Result /= Void
            end
```

Table 14: Feature *cw_argument*: Generating a *Contract Wizard* argument

The feature *cw_argument* creates a *Contract Wizard* argument with a *Contract Wizard* argument name (*CW_NAME*) (1). To do this, it first creates a *Contract Wizard* type (*CW_TYPE*) representing the type of the *.NET* argument and sets it to the *Contract Wizard* type (2).

4.4.5 The feature *cw_procedure*

```
cw_procedure (a_procedure: METHOD_INFO): CW_PROCEDURE is (1)
                -- Contract Wizard procedure (AST node) from .NET procedure
                -- a_procedure
        require
                procedure_not_void: a_procedure /= Void
        do
                Result := cw_procedure_from_method_base (a_procedure) (2)
        ensure (3)
                result_not_void: Result /= Void
                deferred_set: Result.is_deferred = a_procedure.is_abstract
                static_set: Result.is_static = a_procedure.is_static
        end
```

Table 15: Feature *cw_procedure*: Generating a *Contract Wizard* procedure

The feature *cw_procedure* creates a *Contract Wizard* procedure from a *.NET* procedure (1). It sets the procedure name (*CW_NAME*), the feature *is_deferred* and the feature *is_static* using the feature *cw_procedure_from_method_base* (2). *Is_static* is set to true if the *.NET* procedure (of type *METHOD_INFO* corresponding to the *.NET System.Reflection.MethodInfo*) is static. *Is_deferred* is set to true if the *.NET* procedure is abstract.

The postcondition ensures that features *is_deferred* and *is_static* of *CW_PROCEDURE* are set (3).

4.4.6 The feature *cw_function*

```
cw_function (a_function: METHOD_INFO): CW_FUNCTION is (1)
                -- Contract Wizard function (AST node representing a .NET
                -- function) created from .NET function a_function
        require
                a_function_not_void: a_function /=Void
                a_function_name_not_void: a_function.name /= Void
```

```
        local
                a_procedure: CW_PROCEDURE
                a_return_type: CW_TYPE
        do
                a_return_type := cw_type (a_function.return_type)
                a_procedure := cw_procedure_from_method_base (a_function) (2)
                create Result.make_from_procedure (a_procedure, a_return_type)
                Result.set_is_property (is_property (a_function)) (3)
        ensure (4)
                result_not_void: Result /= Void
                deferred_set: Result.is_deferred = a_function.is_abstract
                property_set: Result.is_property = is_property (a_function)
                static_set: Result.is_static = a_function.is_static
        end
```

Table 16: Feature *cw_function*: Generating a *Contract Wizard* function

The feature *cw_function* creates a *Contract Wizard* function from a *.NET* function (1). It sets
the procedure name (*CW_NAME*), the feature *is_deferred* and the features *is_static* using the
feature *cw_procedure_from_method_base* (2). *Is_static* is set to true if the parsed procedure
(of type *METHOD_INFO* corresponding to the *.NET System.Reflection.MethodInfo*) is static.
Is_deferred is set to true if the parsed function is abstract.

It sets the feature *is_property* (3). The feature *is_property* is set to true if the parsed function
is an *accessor* method (for example "get_property").

The postcondition of *cw_function* ensures that features of *CW_FUNCTION* are set (4).

4.4.7   The feature *cw_attribute*

```
        cw_attribute (a_field: FIELD_INFO): CW_ATTRIBUTE is (1)
                -- Contract Wizard attribute from .NET field a_field
        require
                a_field_not_void: a_field /= Void
                a_field_name_not_void: a_field.name /= Void
        local
                an_attribute_name: CW_NAME
                a_type: CW_TYPE
                a_value: STRING
        do
                an_attribute_name := cw_feature_name (a_field.name.to_string)
                create Result.make (an_attribute_name) (2)
                a_type := cw_type (a_field.field_type)
                Result.set_type (a_type) (3)
                Result.set_static (a_field.is_static) (4)
                Result.set_constant (a_field.is_literal)
```

```
                    if a_field.is_literal then
                        create a_value.make_from_cil (a_field.get_value
                            (a_field.reflected_type).to_string)
                        Result.set_value (a_value)
                    end
            ensure (5)
                result_not_void: Result /= Void
                static_set: Result.is_static = a_field.is_static
                constant_set: Result.is_constant = a_field.is_literal
                constant_value_set: Result.is_constant
                        implies Result.value /= Void
            end
```

Table 17: Feature *cw_attribute*: Generating a *Contract Wizard* attribute

The feature *cw_attribute* creates a *Contract Wizard* attribute (*CW_ATTRIBUTE*) from a *.NET* field (1). First, it creates the *Contract Wizard* attribute with the attribute name (*CW_NAME*) (2). Then, it creates a *Contract Wizard* type (*CW_TYPE*) from the field type and sets it to the *Contract Wizard* attribute (3). It also sets the features *is_static* and *is_constant* depending on the parsed field (4). The postcondition ensures that the features of *CW_ATTRIBUTE* are set (5).

4.4.8 The feature *cw_creation_procedure*

```
        cw_creation_procedure (a_constructor: CONSTRUCTOR_INFO):
            CW_CREATION_PROCEDURE is (1)
                -- Contract Wizard creation procedure from .NET constructor
                -- a_constructor
            require
                a_constructor_not_void: a_constructor /= Void
            local
                a_procedure: CW_PROCEDURE
            do
                a_procedure := cw_procedure_from_method_base (a_constructor)
                create Result.make_from_procedure (a_procedure)
            ensure
                result_not_void: Result /= Void
                result_is_not_static: Result.is_static = False
                result_is_not_deferred: Result.is_deferred = False
            end
```

Table 18: Feature *cw_creation*_procedure: Generating a *Contract Wizard* creation procedure

The feature *cw_creation_procedure* creates a *Contract Wizard* creation procedure (*CW_CREATION_PROCEDURE*) from a *.NET* constructor (1). The creation procedure must be neither static nor deferred. It is guaranteed by the postconditions: *result_is_not_static* and *result_is_not_deferred* (2).
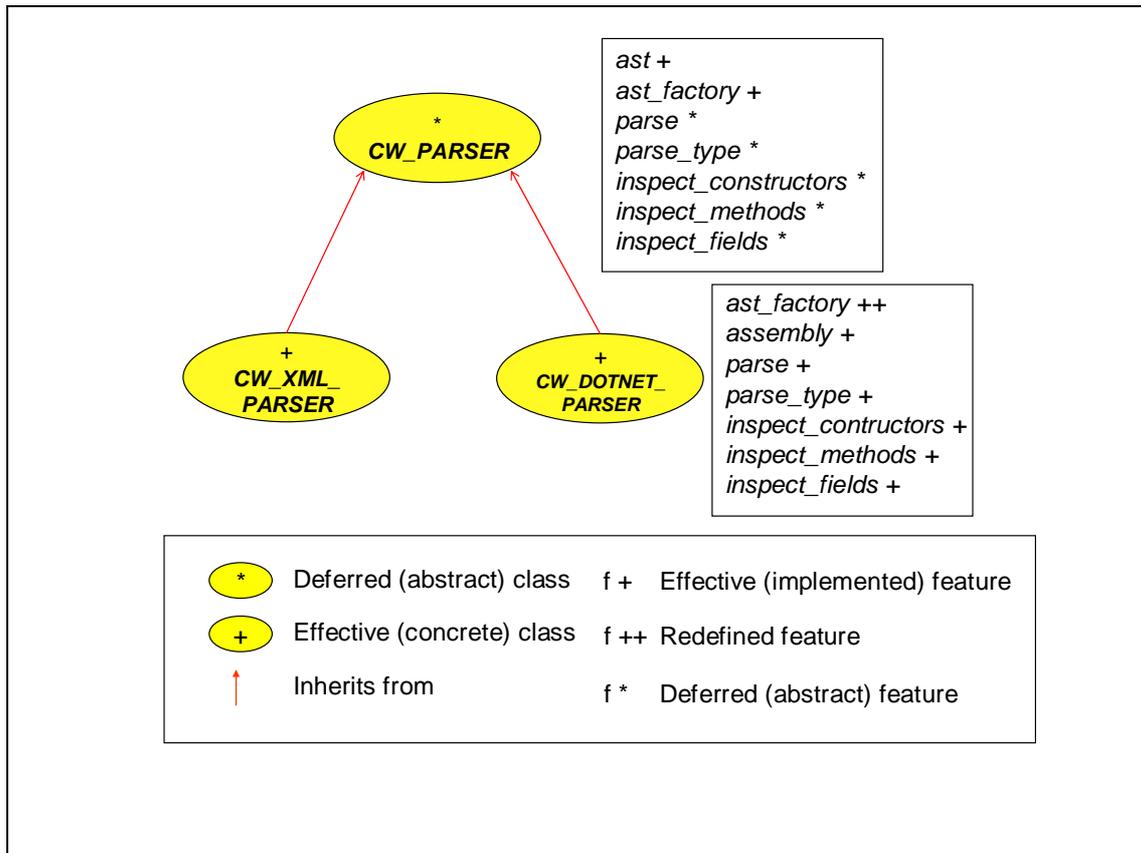
42

4.5 The parser (*CW_PARSER*)

Figure 13: Parser hierarchy

The parser consists of the following classes: *CW_PARSER, CW_DOTNET_PARSER* and *CW_XML_PARSER* (see *Figure 13*).

The class *CW_PARSER* defines and partly implements the features common for *CW_XML_PARSER* and *CW_DOTNET_PARSER*.

The class *CW_DOTNET_PARSER* parses a *CLS* compliant *.NET* assembly and generates the data structure *ast* (of type *LINKED_LIST* [*AST_NODE*]). The data structure *ast* contains all required information to generate the proxy classes.

The class *CW_XML_PARSER* parses the *XML* representation of the data structure *ast* and generates the object representation of the data structure *ast*.

The class *CW_PARSER* implements the following attributes: *ast* and *ast_factory* (see *Figure 13*). The data structures *ast* and *ast_factory* are described above.

The class *CW_PARSER* defines the following deferred features: *parse*, *parse_type*, *inspect_constructors*, *inspect_fields*, *inspect_methods*, *parse_public_constructor*, *parse_public_field* and *parse_public_function* (see *Figure 13*). I will now describe them in more details.

- **Feature** *parse*
  It parses all types by calling the feature *parse_type* for every type and generates the data structure *ast* (of type *LINKED_LIST* [*CW_TYPE*]).

- **Feature** *parse_type*
  It parses a type by using features: *inspect_constructors, inspect_fields, inspect_methods* and creates a *Contract Wizard* type (*CW_TYPE*). It adds the created *Contract Wizard* type to the *ast.*

- **Feature** *inspect_constructors*
  This feature iterates through all constructors and calls the feature *parse_public_constructor* for every inspected constructor.

- **Feature** *inspect_fields*
  This feature iterates through all fields and calls the feature *parse_public_field* for every inspected field.

- **Feature** *inspect_methods*
  This feature iterates through all methods and calls the feature *parse_public_function* or *parse_public_procedure* for every method depending on the return type.

- **Feature** *parse_public_constructor*
  It parses a constructor and generates a *Contract Wizard* creation procedure (*CW_CREATION_PROCEDURE*). It adds the created object to the *Contract Wizard* type (*CW_TYPE*).

- **Feature** *parse_public_field*
  It parses a field and generates a *Contract Wizard* attribute (*CW_ATTRIBUTE*). It adds the created object to the *Contract Wizard* type (*CW_TYPE*).

- **Feature** *parse_public_function*
  It parses a public function and produces a *Contract Wizard* function (*CW_FUNCTION*). It adds the created object to the *Contract Wizard* type (*CW_TYPE*).

4.6 The .*NET* parser (*CW_DOTNET_PARSER*)

The class *CW_DOTNET_PARSER* gathers information needed to generate proxy classes. It provides the following features: *make*, *parse*, *parse_type*, *inspect_methods*, *inspect_fields*, *inspect_constructors*, *parse_public_constructor*, *parse_public_function*, *parse_public_procedure* and *parse_public_field* (see *Figure 13* above). I will now describe them in more detail.

44

4.6.1 The creation procedure

```
make (an_assembly: like assembly) is (1)
            -- Initialize current with a CLS compliant an_assembly.
            -- Generate ast.
    require
            assembly_not_void: an_assembly /= Void
            assembly_is_cls_compliant: is_cls_compliant (an_assembly) (2)
    do
            -- Create an empty ast.
            create ast.make (3)
            -- Create ast factory to enable generation of cw ast nodes.
            create ast_factory.make (3)
            assembly := an_assembly
    ensure
            assembly_assigned: assembly = an_assembly
    end
```

Table 19: Creation procedure of the .NET parser

This feature initializes the parser with a .*NET* assembly that has to be parsed (1).
The .*NET* assembly must be *CLS* compliant; hence the precondition
assembly_is_cls_compliant (2). Moreover, this feature initializes the data structure *ast* and the
ast factory (3). This is ensured by the class invariant (clauses *ast_not_void* and
ast_factory_not_void).

4.6.2 Feature *parse*

```
parse is
            -- Parse all public types from assembly.
    local
            i: INTEGER
    do
            types := assembly.get_types
            from
                    i := types.lower
            until
                    i = types.count
            loop
                    type := types.item (i)
                    if type_exists (2) then
                            parse_type (1)
                    end
                    i := i + 1
            end
```

```
        ensure then
                types_not_void: types /= Void
                all_types_parsed: type = types.item (types.count-1)
                consistent: ast.count = type_count
        end
```

Table 20: Feature *parse*: Parsing public types

This feature iterates through all *types* of a *.NET* assembly (*ASSEMBLY*) and calls *parse_type* (1) for every inspected public type (2).

4.6.3 Parsing a type

```
parse_type is
            -- Parse a public type and all its members like method_infos,
            -- field_infos, constructor_infos.
            -- Generate a cw_type with all its features:
            -- cw_procedure, cw_function, cw_attribute.
        do
            cw_type := ast_factory.cw_type (type) (1)
            if has_public_constructors then
                    inspect_constructors (2)
            end
            if has_public_methods then
                    inspect_methods (3)
            end
            if has_public_fields then
                    inspect_fields (4)
            end
            ast.extend (cw_type) (5)
        end
```

Table 21: Feature *parse_type*: Parsing public members of a type

The feature *parse_type* creates a *Contract Wizard* type (*CW_TYPE*) from the *.NET type* being parsed. The *.NET ast* factory (*CW_AST_FACTORY*) facilitates the creation of the *Contract Wizard* type (*CW_TYPE*). It sets all required features: *is_static, is_deferred, is_expanded* and *is_enum* (1).

It parses the public constructors, methods and fields (members) and calls the features *inspect_constructors* (2), *inspect_methods* (3) and *inspect_fields* (4) depending on the type of the parsed member. The feature *inspect_constructors* iterates through every constructor of the *.NET* type and calls the feature responsible for parsing it (*parse_public_constructor*). The feature *inspect_methods* does the same for *.NET* methods. The feature *inspect_fields* takes care of *.NET* fields.

It adds the resulting *Contract Wizard* type to the *ast* (5).

46

4.6.4 Inspect methods

```
inspect_methods is
        -- Inspect all public method_infos from type
        -- Call parse_public_procedure or parse_public_function for every
        -- inspected method_info depending on the method_info type.
    local
        i: INTEGER
    do
        method_infos := type.get_methods
            -- Inspect all public method_infos from type.
        from
            i := method_infos.lower
        until
            i = method_infos.count
        loop
            method_info := method_infos.item (i)
            if procedure_exists then
                parse_public_procedure (1)
            elseif function_exists then
                parse_public_function (2)
            end
            i := i + 1
        end
    ensure then
        all_methods_parsed: method_info = method_infos.item
            (method_infos.count - 1)
    end
```

Table 22: Feature *inspect_methods*

The feature *inspect_methods* iterates through all public methods of the *.NET* type being parsed. It calls the feature *parse_public_method* (1) or *parse_public_function* (2) for every inspected method depending on the return type of the *.NET* method.

4.6.5 Inspect fields

The feature *inspect_fields* iterates through all public fields of the parsed *.NET* type. It calls the feature *parse_public_field* for every inspected public field. (The code resembles the code fragment in *Table 22*.)

4.6.6 Inspect constructors

The feature *inspect_contructors* iterates through all public constructors of the parsed *.NET* type. It calls the feature *parse_public_constructor* for every inspected public constructor. (The code resembles the code fragment in *Table 22*.)

4.6.7 Parse constructors

```
parse_public_constructor is
            -- Parse a constructor_info.
            -- Generate corresponding cw_creation_procedure.
            -- Add cw_creation_procedure to cw_type.
        do
            cw_creation_procedure := ast_factory.cw_creation_procedure
                (constructor_info) (1)
            cw_type.add_feature (cw_creation_procedure) (2)
        end
```

Table 23: Feature *parse_public_constructor*

The feature *parse_public_constructor* parses a public *.NET* constructor (class *CONSTRUCTOR_INFO* corresponding to the *.NET System.Reflection.ConstructorInfo*) and generates a *Contract Wizard* creation procedure (*CW_CREATION_PROCEDURE*) from it. The *.NET ast* factory (*CW_DOTNET_AST_FACTORY*) facilitates the creation of the *Contract Wizard* creation procedure (*CW_CREATION_PROCEDURE*) (1). It sets all required features: *is_static* and *is_deferred*. The generated creation procedure is added to the feature list of the *Contract Wizard* type (*CW_TYPE*) (2).

4.6.8 Parse functions

The feature *parse_public_function* parses a public function and generates a *Contract Wizard* function (*CW_FUNCTION*). The *.NET ast* factory facilitates the creation of the *Contract Wizard* function. It sets all required features: *is_static* and *is_deferred*. The generated function is inserted to the feature list of the *Contract Wizard* type. (The code resembles the code fragment in *Table 23*.)

4.6.9 Parse procedures

The feature *parse_public_procedure* parses a public procedure and generates a *Contract Wizard* procedure (*CW_PROCEDURE*). The *.NET ast* factory facilitates the creation of the *Contract Wizard* procedure. It sets all required features: *is_static, is_deferred* and *is_property*. The generated function is added to the feature list of the *Contract Wizard* type. (The code resembles the code fragment in *Table 23*.)

4.6.10 Parse fields

The feature *parse_field* parses a public field and generates a *Contract Wizard* attribute (*CW_ATTRIBUTE*) from it. The *.NET ast* factory facilitates the creation of a *Contract Wizard* attribute. It sets all required features: *is_static, is_constant* and *value*. The feature *value* is set only if the field is a constant. The generated attribute is inserted to the feature list of the *Contract Wizard* type. (The code resembles the code fragment in *Table 23*.)
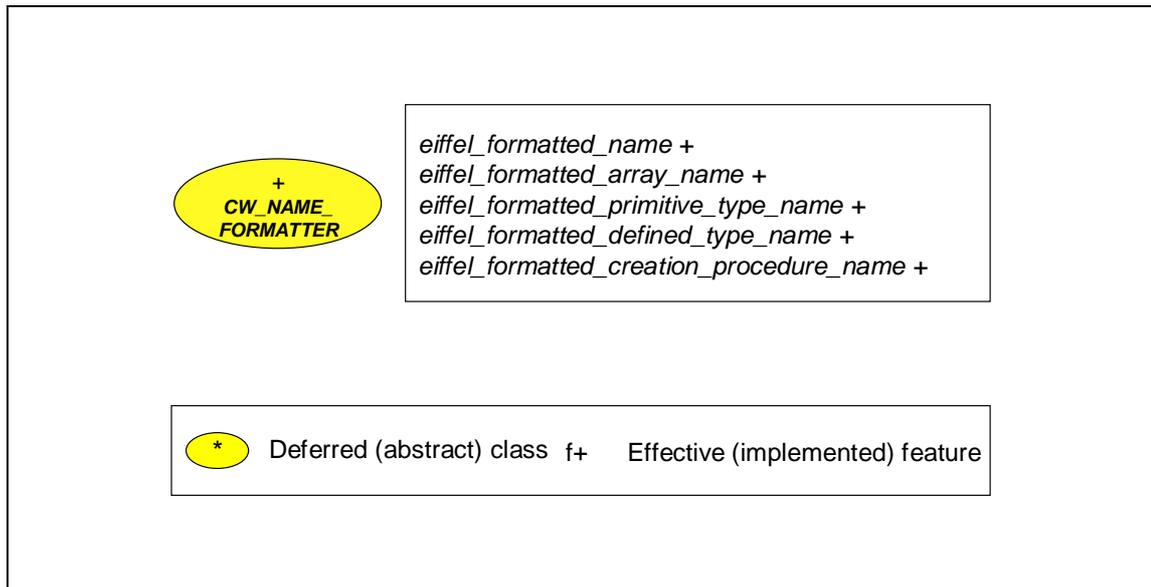
4.7 Translating .*NET* names into *Eiffel* names

Figure 14: Name formatter: Transforming .*NET* names into *Eiffel* names

Since *Eiffel* names and .*NET* names differ slightly, we need a class that translates .*NET* names into *Eiffel* names. The class *CW_NAME_FORMATTER* is responsible for this. It implements the following features: *eiffel_formatted_name, eiffel_formatted_keyword_name, eiffel_formatted_array_name* and *eiffel_formatted_primitive_type_name* (see *Figure 14* above). I now describe them in more detail.

- **Feature** *eiffel_formatted_name*
  By convention, feature names in Eiffel use all lower case characters, and like class names, words are separated by underscore [11]. The feature *eiffel_formatted_name* translates "CamelCase" .*NET* names into *Eiffel* names separated by underscore. For example, "getName" becomes "get_name".

- **Feature** *eiffel_formatted_keyword_name*
  If the name corresponds to an Eiffel keyword an underscore is append to it: For example the following .*NET* member names: "*feature*", "*is*", "*do*" become "*feature_*", "*is_*", "*do_*". All keywords are defined in the class *CW_EIFFEL_KEYWORD_CONSTANTS.*

- **Feature** *eiffel_formatted_array_name*
  Arrays have the following format in .*NET*: "*type* [,] *name*". Example: "*string* [,] *arrayName*". The commas stand for the dimension of the array. If the array has dimension one there are no commas. If array has dimension i and i is bigger than 1 there are i-1 commas, between the brackets.
  This feature translates a .*NET* array into an *Eiffel* array. The class *NATIVE_ARRAY* represents a .*NET* array in *Eiffel for .NET*. Arrays are written as: "*NATIVE_ARRAY* [*ARRAY_TYPE*]" independently of the dimension. The above example is translated into "*NATIVE_ARRAY* [*STRING*]".

49

- **Feature** *eiffel_formatted_primitive_type_name*
  The *.NET* Framework has the following *CLS* compliant primitive types: *byte, int16, int32, int64, single, double, boolean, char, decimal, intPtr,* and *string.* The feature *eiffel_formatted_primitive_type_name* translates primitive *CLS* compliant *.NET* types into *Eiffel* according to the following translation rule:

Byte → INTEGER_8
Int16 → INTEGER_16
Int32 → INTEGER
Int64 → INTEGER_64
Single → REAL
Double → DOUBLE
Boolean → BOOLEAN
Char → CHARACTER
Decimal → DECIMAL
Object → SYSTEM_OBJECT
IntPtr → POINTER
String → SYSTEM_STRING

  Table 24: Translation rule for primitive types

- **Feature** *eiffel_formatted_defined_type_name*
  Eiffel class names must be unique within a system. There are *.NET* class names that conflict with the class names in the *EiffelBase* library. They are renamed according to the following translation rule:

Object → SYSTEM_OBJECT
String → SYSTEM_STRING
Array → SYSTEM_ARRAY
Console → SYSTEM_CONSOLE
DateTime → SYSTEM_DATE_TIME
Directory → SYSTEM_DIRECTORY
File → SYSTEM_FILE
Queue → SYSTEM_QUEUE
Random → SYSTEM_RANDOM
SortedList → SYSTEM_SORTED_LIST
Stack → SYSTEM_STACK
Stream → SYSTEM_STREAM

  Table 25: Translation rule for types already defined in the *EiffelBase* library

- **Feature** *eiffel_formatted_creation_procedure_name*
  The name of the constructor in *.NET* is *.ctor*. This feature renames it to *make.*

4.8 Summary

4.8.1 Characteristics of the *.NET* parser

- The assembly being parsed has to be *CLS* compliant.
- The parser parses the following information required to generate *Eiffel* proxy classes:
  - o public types (class *TYPE* corresponding to .*NET System.Type*)
  - o public .NET constructors (class *CONSTRUCTOR_INFO* corresponding to .*NET System.Reflection.ConstructorInfo*)
  - o public .NET methods (class *METHOD_INFO* corresponding to .*NET System.Reflection.MethodInfo*)
  - o public .NET fields (class *FIELD_INFO* corresponding to .*NET System.Reflection.FieldInfo*)

- The parser generates the following *Contract Wizard* nodes:
  - o The parser generates a Contract Wizard type (*CW_TYPE*) and sets the following features for it: *is_deferred*, *is_array*, *is_expanded* and *is_enum*, *name_space*. Moreover, it attaches all interfaces (of type *CW_INTERFACE*) that the type implements.
  - o The parser generates a Contract Wizard attribute (*CW_ATTRIBUTE*) and sets the following features for it: *is_static*, *is_constant* and if the parsed field is a constant it also sets the constant value.
  - o The parser generates a Contract Wizard procedure (*CW_PROCEDURE*) and sets the following features for it: *is_static*, *is_deferred*.
  - o The parser generates a Contract Wizard function (*CW_FUNCTION*) and sets the following features for it: *is_static*, *is_deferred* and *is_property* (is set to true if the function is an accessor).
  - o The parser generates a Contract Wizard creation procedure (*CW_CREATION_PROCEDURE*).

The *CW_NAME_FORMATTER*
- translates "CamelCase" .*NET* names into *Eiffel* names separated by underscore.
- keeps track of the following *CLS* compliant primitive types: *byte*, *int16*, *int32*, *int64*, *single*, *double*, *boolean*, *char*, *decimal*, *intPtr* and *string*.
- keeps track of .*NET* class names that conflict with the *EiffelBase* library.
- keeps track of .*NET* names that conflict with keywords of the *Eiffel* language (For example: *feature, do, end, ...*).
- handles arrays of dimension n.

4.8.2 Limitations

- The parser does not parse events (class *EVENT_INFO*, System.ReflectionEventInfo in .*NET*), since the event mechanism of .*NET* differs from the agent mechanism of *Eiffel*.
- The parser does not parse nested classes.

5. The XML parser (Contract Reader)

This chapter discusses the implementation of the *XML* parser. First, I evaluate two parsers and explain why I selected the *Gobo XML* parser. Then, I describe the *XML* parser I developed. Finally, I discuss the event handling mechanism.

5.1 Design of the *XML* parser

5.1.1 Evaluating the right parser

There are quite a few libraries to parse *XML* files in *Eiffel for .NET*. I have evaluated the two following libraries:

- *Gobo*, a free and portable library for *Eiffel* [10]
- *XML* libraries part of the *.NET* Framework [14]

The *Gobo* library contains two non-validating *XML* parsers: A pure *Eiffel* parser and a parser based on the *expat C library* [13].

The *XML* libraries part of the *Eiffel .NET* Framework contain a variety of validating and non-validating parsers.

I decided to use the *Eiffel* parser. The main reason is that the *Eiffel* parser unlike its competitors is written in pure *Eiffel*. Moreover, it is fast and simple to use.

Even if the *Eiffel* parser is a non-validating parser, I wrote a document type definition. It can be very useful to test the generated *XML* files. The *XML* files can be validated using external tools, such as *xmlspy* [15].

The *Eiffel* parser is an event parser. It reads an *XML* stream sequentially and acts as an event source. It throws events on *XML* start tags, *XML* end tags and *XML* attributes.

The *Eiffel* parser is initiated by the class *CW_XML_PARSER* described below.

5.2 Structure of the *XML* parser
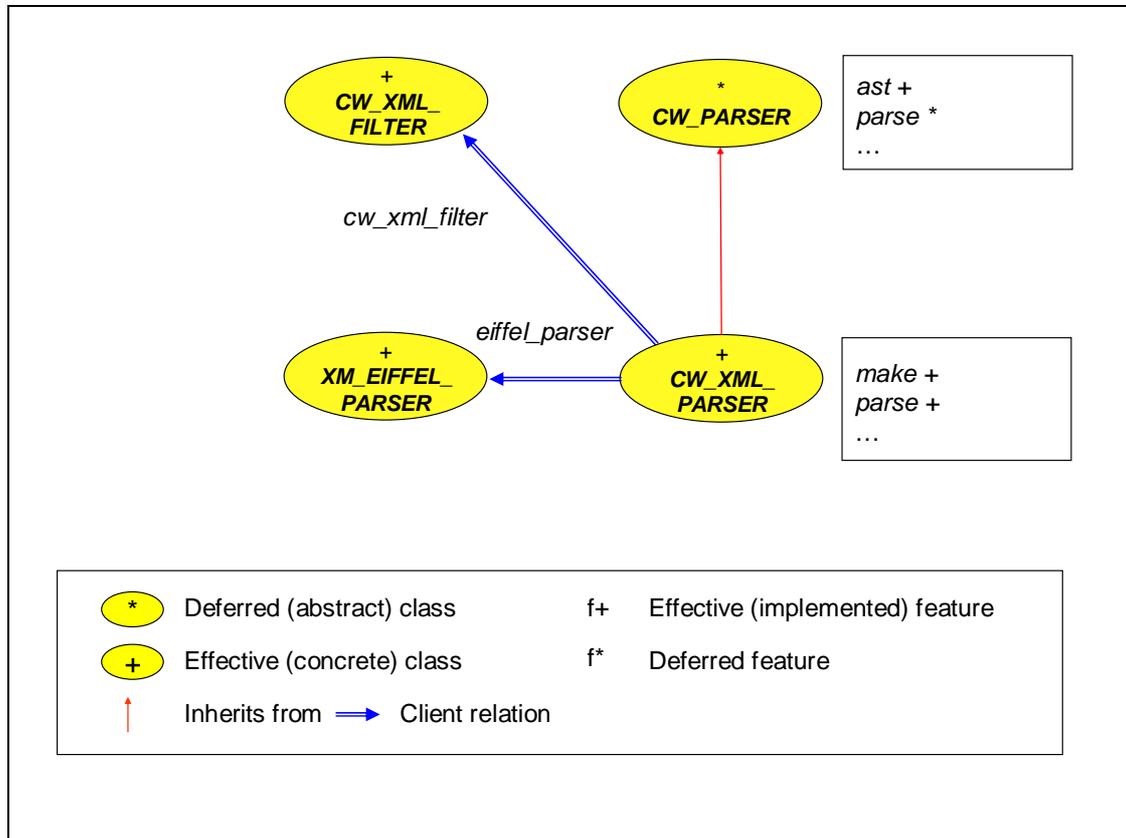
5.2.1   Class hierarchy

Figure 15: Class hierarchy of the XML parser

The parser consists mainly of the following classes: *CW_PARSER*, *CW_XML_PARSER*, *CW_XML_FILTER* and *XM_EIFFEL_PARSER* (see *Figure 15* above). *XM_EIFFEL_PARSER* is part of the *Gobo* library. I wrote the other classes. The classes with prefix *CW_* are part of the *Contract Wizard*.

5.2.2   The class *CW_PARSER*

The class *CW_PARSER* defines the feature *parse* implemented by the class *CW_XML_PARSER* and exposes the attribute *ast* (of type *LINKED_LIST* [like *cw_type*]) where *cw_type* represents the *Contract Wizard* type being parsed (see *Figure 15* above).

5.2.3   The classes *CW_XML_FILTER* and *XM_EIFFEL_PARSER*

The class *CW_XML_FILTER* acts as an event handler and the class *XM_EIFFEL_PARSER* acts as an event source. (*XM_EIFFEL_PARSER* is part of the *Gobo* library.)

5.2.4 The class *CW_XML_PARSER*

The role of the class *CW_XML_PARSER* is to parse the *XML* representation of a *Contract Wizard* proxy class using the *Eiffel* parser (*XM_EIFFEL_PARSER*) and to generate the data structure *ast* from it.

The class *CW_XML_PARSER* implements the following features: the initialization feature *make* and the feature *parse* (*see Figure 15* above).

The initialization procedure *make* initializes the *file_name* and creates an empty *ast* (of type *LINKED_LIST* [like *cw_type*]). The filename (*file_name*) represents the name of the *XML* file.

5.3 Setting up the *GOBO XML* parser

```
parse is
            -- Parse an XML file with name file_name containing the
            -- serialized proxy classes.
            -- Create ast.
     do
            create file.make (file_name) (1)
            file.open_read
            if not valid_file then
                    -- Error
            else
                    create eiffel_parser.make (2)
                    create cw_xml_filter.make (ast) (3)
                    eiffel_parser.set_callbacks (cw_xml_filter) (4)
                    eiffel_parser.parse_from_stream (a_file) (5)
                    file.close
            end
     ensure
            xml_filter_not_void: valid_file implies cw_xml_filter /= Void
            eiffel_parser_not_void: valid_file implies eiffel_parser /= Void
            eiffel_parser_is_correct: valid_file implies eiffel_parser.is_correct
     end
```

Table 26: Feature *parse*

The feature *parse* (see *Table 26* and *Figure 15* above) creates a file (1), creates the *XML* parser (*XM_EIFFEL_PARSER*) (2) and instantiates the events filter (*CW_XML_FILTER*) (3).

The feature *set_callbacks* registers the event filter (*CW_XML_FILTER*) to the *XML* parser (*XM_EIFFEL_PARSER*) (4).

5.3.1 The parsing process

The feature *parse_from_stream* (5) initiates the parsing process. The parser reads the *XML* stream sequentially and throws events on *XML* tags. They are handled by the registered event filter (*CW_XML_FILTER*) described in the next chapter.

5.3.2 The event filter

The event filter consists of the following classes: *CW_XML_FILTER, CW_XML_CONSTANTS* and *XM_CALLBACKS_FILTER* (see *Figure 16* below).

The class *CW_XML_FILTER*
The class *CW_XML_FILTER* redefines the following features from the class *XM_CALLBACKS_FILTER*: *on_start_tag*, *on_attribute*, *on_start_tag_finish* and *on_end_tag* in order to add new functionality.

The class *XM_CALL_BACK_FILTER*
The class *XM_CALL_BACK_FILTER* makes the class *CW_XML_FILTER* acting as an event handler.

The class *CW_XML_CONSTANTS*
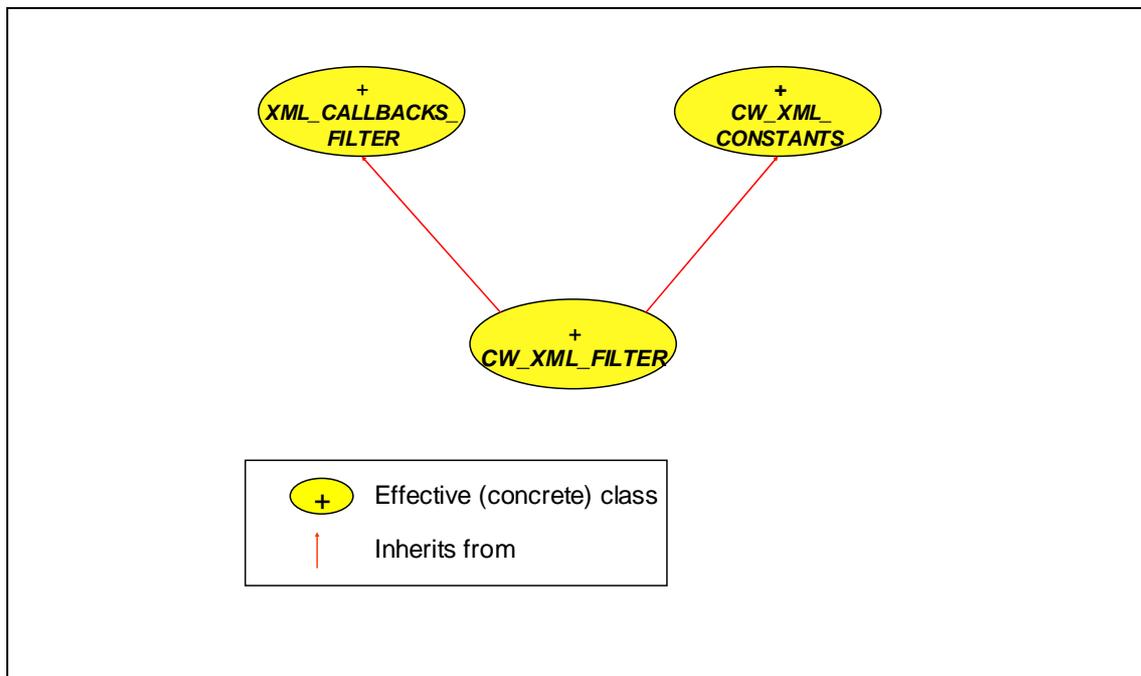The class *CW_XML_CONSTANTS* provides the *CW_XML_FILTER* class with constants.

Figure 16: Class hierarchy of the event filter

5.3.3 Cooperation between *CW_XML_PARSER* and *CW_XML_FILTER*

I will explain the cooperation between the *CW_XML_PARSER* (event source) and the *CW_XML_FILTER* (event handler) on the example in *Table 27* below.

The example demonstrates the serialized content of a *Contract Wizard* proxy class in *XML*. It shows a list of *Contract Wizard* types. The first type has the name *CLASS*. It contains a creation procedure *make* with one argument and a procedure *foo*.

```
<cw_ast>
    <cw_type dotnet_name = "class" eiffel_name = "CLASS"        >
        <cw_creation_procedures>
            <cw_creation_procedure …eiffel_name="make" …>
                <cw_arguments>
                    <cw_argument dotnet_name="…"…      >
                        <type … is_array = "no" />
                    </cw_argument>
                </cw_arguments>
            </cw_creation_procedure>
        </cw_creation_procedures>
        <cw_procedures >
            <cw_procedure … eiffel_name= "foo" …>
                …
    </cw_type>
    <cw_type>
…
</cw_ast>
```

Table 27: Example of a Contract Wizard XML file

5.3.4 Event handling

As mentioned before the *XML* parser fires events on *XML* tags handled by the following features of class *CW_XML_FILTER*: *on_start_tag, on_start_tag_finish, on_attribute* and *on_end_tag*.
The feature *on_start_tag* handles events fired when an *XML* start tag has been processed. The feature *on_start_tag_finish* handles events when the end of an *XML* start tag has been processed. The feature *on_attribute* handles events fired when an *XML* attribute has been processed and finally the feature *on_end_tag* handles events fired when an *XML* end tag has been processed.

5.3.5 Parse XML

A closer look at a *Contract Wizard XML* start tag shows that there is enough information stored in it to create a *Contract Wizard ast* node (see *Table 27*). The parser creates a *Contract Wizard ast* node for each *XML* start tag using the feature *on_start_tag_finish* (see *Table 28* below).

The specific type (*CW_PROCEDURE, CW_FUNCTION, and CW_ATTRIBUTE*) of the *Contract Wizard ast* node depends on the name of the *XML* start tag.

In *Table 28* below, you see the implementation of the feature *on_start_tag_finish*. The *XML* node (represented by *cw_current*) is queried in the *"if statements"*. Depending on the name of the *XML* node, the following features are called: *generate_type, generate_procedure, generate_precondition ….*

```
on_start_tag_finish is
        -- Generate Contract Wizard ast nodes.
    do
        if is_type (cw_current) then
            generate_type
        elseif is_attribute (cw_current) then
            generate_attribute
        elseif is_creation_procedure (cw_current) then
            generate_creation_procedure
        elseif is_procedure (cw_current) then
            generate_procedure
        elseif is_function (cw_current) then
            generate_function
        elseif is_feature_type (cw_current) then
            generate_return_type
        elseif is_argument (cw_current) then
            generate_argument
        elseif is_precondition (cw_current) then
            generate_precondition
        elseif is_postcondition (cw_current) then
            generate_postcondition
        elseif is_invariant (cw_current) then
            generate_invariant
        end
        Precursor {XM_CALLBACKS_FILTER}
    end
```

Table 28: Feature *on_start_tag_finish*

Composing *Contract Wizard ast* nodes
In the previous section, I discussed how to create *Contract Wizard ast* nodes.
The next step is to compose them. All *Contract Wizard ast* nodes defined between an *XML* start tag and an *XML* end tag have to be composed.

For example if the *XML* tag represents a *Contract Wizard* type the parser assigns all *Contract Wizard* features defined between the *XML* start tag (<cw_type>) and the *XML* end tag (</cw_type>) to this *Contract Wizard* type .

The feature *on_end_tag* (see *Table 29* below) is responsible for composing the *Contract Wizard ast* nodes (1), (2), (3)… (It is called by the parser whenever an *XML* end tag has been processed).

```
on_end_tag (a_namespace: STRING; a_prefix: STRING;
      a_local_part: STRING) is
            -- Compose Contract Wizard AST nodes.
      local
            upper_part: STRING
      do
            upper_part := a_local_part.as_upper
            if is_type (upper_part) then
                  compose_type (1)
                  extend_ast
            elseif is_creation_procedure (upper_part) then
                  compose_routine (cw_creation_procedure) (2)
            elseif is_procedure (upper_part) then
                  compose_routine (cw_procedure) (3)
            elseif is_function (upper_part) then
                  compose_routine (cw_function) (4)
                  cw_function.set_return_type (return_type)
            elseif is_attribute (upper_part) then
                  compose_attribute (5)
            elseif is_argument (upper_part) then
                  compose_argument (6)
            end
            Precursor {XM_CALLBACKS_FILTER} (a_namespace, a_prefix,
                  a_local_part)
      ensure then
            return_type_set: is_function (a_local_part.as_upper) implies
                  cw_function.return_type = return_type
      end
```

Table 29: Feature *on_end_tag*

5.3.6 The document type definition

The document type definition (*DTD*) describes the structure of an *XML* file and validates it. The document type definition in *Table 30* below validates every *Contract Wizard XML* file. You can use the *DTD* with various tools to edit *XML* files such as *xmlspy* [15]

```
<?xml version="1.0" encoding="utf-8"?>
<!ELEMENT cw_ast (cw_type+)>
<!ELEMENT cw_type (interfaces?, cw_creation_procedures?, cw_attributes?,
cw_procedures?, cw_functions?, cw_invariants?)?>
<!ELEMENT cw_creation_procedures (cw_creation_procedure)*>
<!ELEMENT cw_attributes (cw_attribute)*>
```

```
<!ELEMENT cw_procedures (cw_procedure)*>
<!ELEMENT cw_functions (cw_function)*>
<!ELEMENT cw_preconditions (cw_precondition)*>
<!ELEMENT cw_postconditions (cw_postcondition)*>
<!ELEMENT cw_invariants (cw_invariant)*>
<!ELEMENT interfaces (interface)*>
<!ELEMENT cw_arguments (cw_argument)*>
<!ATTLIST cw_type
      dotnet_name CDATA #REQUIRED
      eiffel_name CDATA #REQUIRED
      namespace CDATA #REQUIRED
      is_deferred (yes | no) #REQUIRED
      is_expanded (yes | no) #REQUIRED
      is_enum (yes | no) #REQUIRED
      is_interface (yes | no) #REQUIRED
>
<!ELEMENT cw_creation_procedure (cw_arguments?, cw_preconditions?,
cw_postconditions?)?>
<!ATTLIST cw_creation_procedure
      dotnet_name CDATA #REQUIRED
      eiffel_name CDATA #REQUIRED
>
<!ELEMENT cw_attribute (type+, cw_preconditions?, cw_postconditions?)?>
<!ATTLIST cw_attribute
      dotnet_name CDATA #REQUIRED
      eiffel_name CDATA #REQUIRED
      is_constant (yes | no) #REQUIRED
      value CDATA #REQUIRED
      is_static (yes | no) #REQUIRED
>
<!ELEMENT cw_procedure (cw_arguments?, cw_preconditions?,
cw_postconditions?)>
<!ATTLIST cw_procedure
      dotnet_name CDATA #REQUIRED
      eiffel_name CDATA #REQUIRED
      is_deferred (yes | no) #REQUIRED
      is_static (yes | no) #REQUIRED
>
<!ELEMENT cw_function (cw_arguments?, type+, cw_preconditions?,
cw_postconditions?)>
<!ATTLIST cw_function
      dotnet_name CDATA #REQUIRED
      eiffel_name CDATA #REQUIRED
      is_deferred (yes | no) #REQUIRED
      is_static (yes | no) #REQUIRED
      is_property (yes | no) #REQUIRED
>
```

```
<!ELEMENT cw_argument (type+)>
<!ATTLIST cw_argument
    dotnet_name CDATA #REQUIRED
    eiffel_name CDATA #REQUIRED
>


<!ELEMENT cw_invariant EMPTY>
<!ATTLIST cw_invariant
    tag CDATA #IMPLIED
    expression CDATA #REQUIRED
>
<!ELEMENT cw_precondition EMPTY>
<!ATTLIST cw_precondition
    tag CDATA #IMPLIED
    expression CDATA #REQUIRED
>
<!ELEMENT cw_postcondition EMPTY>
<!ATTLIST cw_postcondition
    tag CDATA #IMPLIED
    expression CDATA #REQUIRED
>
<!ELEMENT interface EMPTY>
<!ATTLIST interface
    dotnet_name CDATA #REQUIRED
    eiffel_name CDATA #REQUIRED
    to_string (yes | no) #REQUIRED
    equals (yes | no) #REQUIRED
    get_hash_code (yes | no) #REQUIRED
>
<!ELEMENT type EMPTY>
<!ATTLIST type
    dotnet_name CDATA #REQUIRED
    eiffel_name CDATA #REQUIRED
    is_array (yes | no) #REQUIRED
>
```

Table 30: The document type definition for the *Contract Wizard* proxy classes
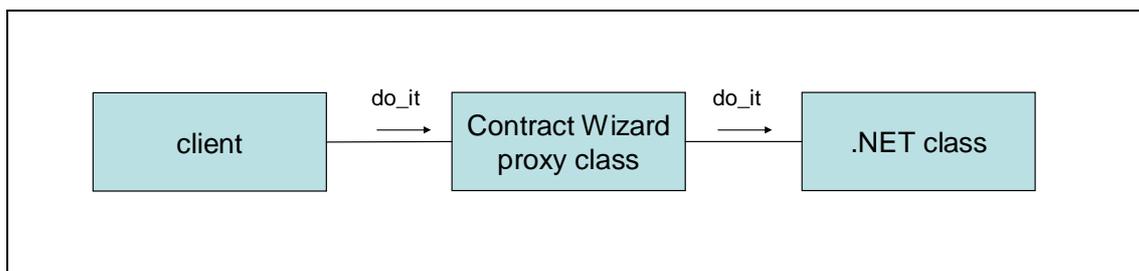
6. Contract Wizard proxy classes

This chapter explains how *Contract Wizard* creates the *Eiffel* representation and the *XML* representation of *Contract Wizard* proxy classes and examines the structure of a *Contract Wizard* proxy class.

6.1 The *Contract Wizard* proxy class

Contract Wizard generates a *Contract Wizard* proxy class for each public *.NET* class or public *.NET* interface in the specified *.NET* assembly.
The *Contract Wizard* proxy class acts as a surrogate for the original *.NET* class or *.NET* interface. To do this the proxy class has to specify the same interfaces and features as the original *.NET* class.

Interactions between the *Contract Wizard* proxy class and the original *.NET* class are as follow:

Figure 17: Delegating calls to the original .NET class

In *Figure 17* you see the interaction between a client, a *Contract Wizard* proxy class and an original *.NET* class. The interaction is the following: the client calls the feature *do_it* of the *Contract Wizard* proxy class instead of calling the feature *do_it* of the original *.NET* class directly. The *Contract Wizard* proxy class delegates the call to the original *.NET* class.
In this way, each call is delegated to the original *.NET* class.
Since the *Contract Wizard* proxy class specifies all public features of the *.NET* class (including inherited features from super classes of the *.NET* class) delegation to inherited features of the *.NET* class is provided.

6.1.1 Implementation of a *Contract Wizard* proxy class

Figure 18 on the next page shows the implementation of a *Contract Wizard* proxy class (*CW_PROXY*).

Every *Contract Wizard* proxy class has a reference *ref* to the original *.NET* class. The reference is of type *ORIGINAL_DOTNET_CLASS* and delegates calls to the original *.NET* class.
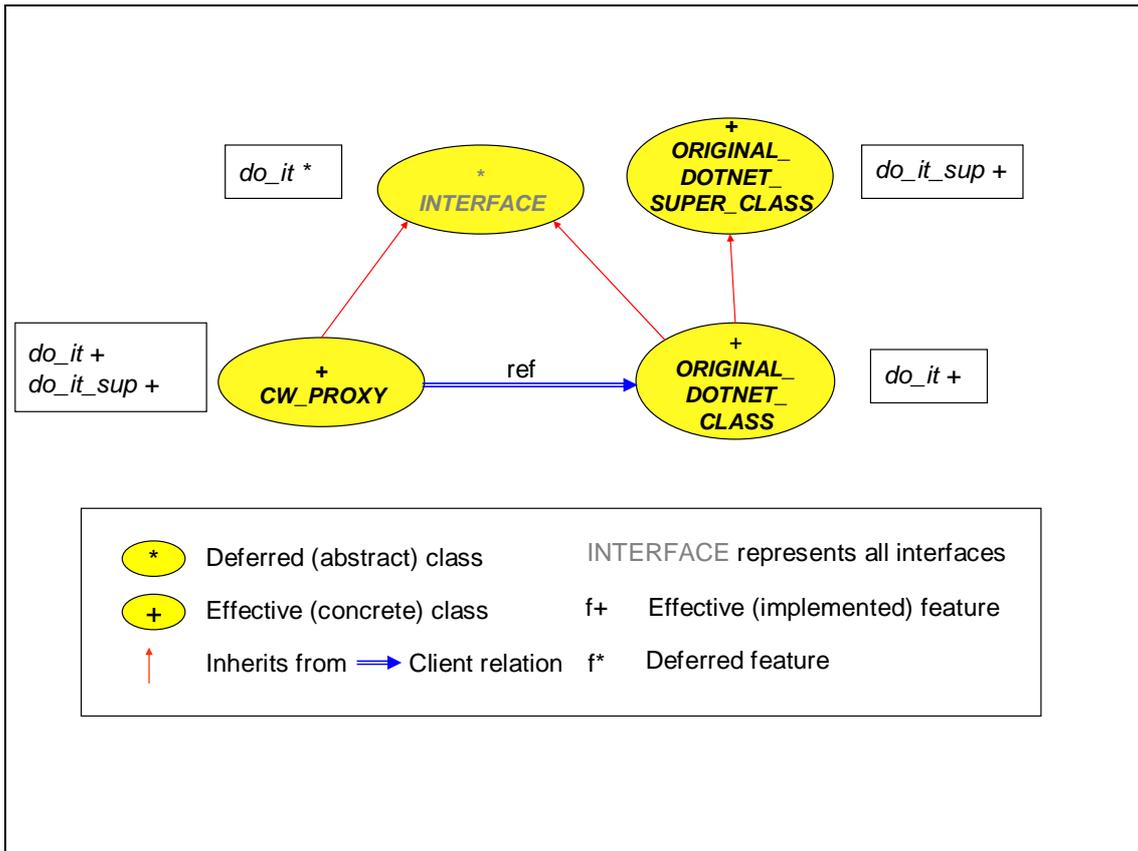
do_it *   *** INTERFACE**   **+ ORIGINAL_ DOTNET_ SUPER_CLASS**   **do_it_sup +**

**do_it +
do_it_sup +**   **+ CW_PROXY**   ref   **+ ORIGINAL_ DOTNET_ CLASS**   **do_it +**

*   Deferred (abstract) class   INTERFACE represents all interfaces
+   Effective (concrete) class   f+   Effective (implemented) feature
↑ Inherits from ⟶ Client relation   f*   Deferred feature

Figure 18: Implementing a Contract Wizard proxy class (*CW_PROXY*)

Both the *Contract Wizard* proxy class (*CW_PROXY*) and the original *.NET* class (*ORIGINAL_DOTNET_CLASS*) implement (inherit from) the same interfaces. The deferred class *INTERFACE* represents all interfaces. That is why it is grey. This structure makes the proxy a real surrogate for the original *.NET* class.

6.1.2   Inheritance from *System.Object*

Every type in .NET is an object, meaning that it must derive directly or indirectly from the Object class. If you do not specify a base class when you define a class, the compiler will inject this requirement into the IL code. [3]
Since *Contract Wizard* proxy classes are also *.NET* classes they inherit from the class *System.Object*.

Before discussing the consequences, I will examine the public methods of the class *System.Object* (in *Eiffel SYSTEM_OBJECT*). From now on I will use the *Eiffel for .NET* naming conventions.

The class *SYSTEM_OBJECT* has the following public methods.
- *equals* (*obj*: *SYSTEM_OBJECT*): *BOOLEAN*: Determines whether the specified *obj* is equal to the current object.

- *frozen equals_object_object (obj_a*: *SYSTEM_OBJECT*; *obj_b*: *SYSTEM_OBJECT)*: *BOOLEAN*: Compares two objects and determines whether they are equal (field by field equality).
- *frozen reference_equals (obj_a*: *SYSTEM_OBJECT*; *obj_b*: *SYSTEM_OBJECT)*: *BOOLEAN*: Compares two object references and determines whether they are referring to the same object (reference equality).
- *get_hash_code*: *INTEGER*: Gets the object hash code.
- *frozen get_type*: *TYPE*: Obtains the object's type at runtime.
- *to_string*: *SYSTEM_STRING*: Gets a string representation of the object.
- *frozen memberwise_clone*: *SYSTEM_OBJECT*: Creates a shallow copy of the current object.

Calls to *equals*, *get_hash_code* and *to_string* are delegated to the original .*NET* class. (The original .*NET* class may provide for example a special hash function).

All other features of the class *SYSTEM_OBJECT* are not delegated. The reason is that they are all *frozen* in the class *SYSTEM_OBJECT*, thus cannot be redefined.

Other reasons are:
- For the feature *get_type*: We want to hide implementation details of the original .*NET* class.
- For the features *equals_object_object, reference_equals, memberwise_clon*e: These features implement the required service and don't affect the current type.

In order to delegate the services of the three features *get_hash_code, equals* and *to_string*, we have to redefine them in the class *SYSTEM_OBJECT*, which every *Contract Wizard* proxy class inherits from. I will explain the delegation of these features on the following three cases:
- the original .*NET* class is an effected class
- the original .*NET* class is an abstract class
- the original .*NET* class is an interface

The original .*NET* class is an effected class
The *Contract Wizard* proxy class has to implement (inherit from) the same interfaces as the original .*NET* class to act as a surrogate.
For example, the *Contract Wizard* proxy class in *Table 31* on the next page generated from the .*NET* core library (mscorlib) [14] implements (inherits from) the following interfaces:
- *IMEMBERSHIP_CONDITION* (1)
- *ISECURITY_ENCODABLE* (1)
- *ISECURITY_POLICY_ ENCODABLE* (1)

These are the same interfaces as the original .*NET* class *ALL_MEMBERSHIP_CONDITIONS* implements.

Handling interfaces
Interface inheritance has some consequences for the *Contract Wizard* proxy class since interfaces are represented by deferred (abstract) classes in *Eiffel for .NET* (abstract classes also inherit from *SYSTEM_OBJECT*), the features *to_string, get_hash_code* and *equals* have

to be undefined to avoid ambiguous feature names. (These features are already inherited from *SYSTEM_OBJECT*). (2)

Interfaces force a programmer to implement defined features. For example, the *.NET* interface *IMEMBERSHIP_CONDITION* (3) forces the programmer to implement (among other features) the features *to_string* and *equals*. This does not affect the *Contract Wizard* proxy class, except the implementation detail that the proxy class needs only undefine the feature *get_hash_code* (4) since the other two features are already defined by the interface.

The class *SYSTEM_OBJECT* has to redefine the three features *to_string*, *equals* and *get_hash_code* to enable delegation of these three features.

```
indexing

        note: "Automatically generated by the Contract Wizard."
        dotnet_name: "System.Security.Policy.AllMembershipCondition"

class CW_ALL_MEMBERSHIP_CONDITION

inherit

        IMEMBERSHIP_CONDITION (1) (3)
                undefine   (4)
                        get_hash_code
                end

        ISECURITY_ENCODABLE (1)
                undefine   (2)
                        get_hash_code,
                        equals,
                        to_string
                end

        ISECURITY_POLICY_ENCODABLE (1)
                undefine (2)
                        get_hash_code,
                        equals,
                        to_string
                end

        SYSTEM_OBJECT
                redefine (5)
                        get_hash_code,
                        equals,
                        to_string
                end
        ....
```

```
        feature {NONE} -- Implementation

                frozen all_membership_condition_ref: ALL_MEMBERSHIP_CONDITION
                        -- Reference to the .NET class

        end
```

Table 31: Fragment of an effected proxy class implementing *.NET* interfaces

The original *.NET* class is an abstract class

From the implementation point of view, there is no difference between effected and deferred *Contract Wizard* proxy classes concerning redefinition.

- Both the effected and the abstract *Contract Wizard* proxy class inherit from *SYSTEM_OBJECT*.
- It is not allowed in *.NET* to specify one of the three features *to_string, equals* or *get_hash_code* as abstract (in *.NET* there is no construct *undefine* like in *Eiffel*).
- This means we can handle the three effected features: *to_string, equals* and *get_hash_code* in the same way we handled them in the effected class.

Overriding

As in the effected *.NET* class a programmer can override one of the three features *to_string, get_hash_code* and *equals* also in the abstract *.NET* class. Overriding has no effect on the current implementation of the *Contract Wizard* proxy class. If a client calls an overridden feature, the call is delegated by the *Contract Wizard* proxy class to the overridden feature of the *.NET* class.

In *Table 32* below, you see an example of a *CLS* compliant *.NET* class written in *C#* and in *Table 33* there is the corresponding *Contract Wizard* proxy class. The feature *ToString* is overridden (1), the feature *Equals* and *GetHashCode* are inherited from *System.Object*.

```
using System;

[assembly: AssemblyVersion ("1.0.3300.1")]
[assembly: AssemblyKeyFile ("mykey.snk")]
[assembly: CLSCompliant (true)]

public abstract class Hello {

        /* Overrides the method ToString from System.Object */
        override public string ToString () { (1)
                return "Hello World!";
        }
}
```

Table 32: *CLS* compliant class Hello overriding the method *ToString* from *System.Object*

Examining the corresponding *Contract Wizard* proxy class in *Table 34* you see that overriding has no effect on the current implementation of a *Contract Wizard* proxy class. The overridden method is called (2) since the reference is of type *HELLO* (1). Note it is not possible to call the method *to_string* from *SYSTEM_OBJECT* through the *Contract Wizard* proxy class in this case.

```
indexing

        note: "Automatically generated by the Contract Wizard."
        dotnet_name: "Hello"

deferred class CW_HELLO

feature -- Query

        frozen get_hash_code: INTEGER is
                    -- dotnet_name: "Hello.GetHashCode (): Int32"
            do
                Result := hello_ref.get_hash_code
            end

        frozen equals (a_obj: SYSTEM_OBJECT): BOOLEAN is
                    -- dotnet_name: "Hello.Equals (obj: Object): Boolean"
            do
                Result := hello_ref.equals (a_obj)
            end

        frozen to_string: SYSTEM_STRING is (2)
                    -- dotnet_name: "Hello.ToString (): String"
            do
                Result := hello_ref.to_string
            end

    feature {NONE}-- Implementation

        frozen hello_ref: HELLO (1)
                    -- Reference to the .NET class

    end
```

Table 33: Deferred proxy class representing the CLS compliant .NET class Hello

The original .*NET* class is an interface
If the original .*NET* class is an interface, the corresponding *Contract Wizard* proxy class is deferred. (*Eiffel* does not support the notion of interface.)
Although the *Contract Wizard* proxy class inherits from *SYSTEM_OBJECT* the features *to_string, equals* and *get_hash_code* have not to be redefined in *SYSTEM_OBJECT*, since a .*NET* interface does not have any effected features by definition.

If a *.NET* interface specifies one of these features the *Contract Wizard* proxy class undefines them in *SYSTEM_OBJECT* as well as all interfaces the *Contract Wizard* proxy class inherits from (see example in *Table 34* below).

Example: The interface *CW_IMEMBERHSIP_CONDITION*
Table 34 represents the *.NET* interface *IMEMBERSHIP_CONDITION* from the *.NET* core library (mscorlib) [14]. The interface specifies the *.NET* members *ToString* and *Equals*. The corresponding *Contract Wizard* proxy class undefines these features in all interfaces (1) it inherits from. These features are also undefined in the class *SYSTEM_OBJECT* (2).

---

indexing

      note: "Automatically generated by the *Contract Wizard*."
      dotnet_name: "System.Security.Policy.IMembershipCondition"

deferred class *CW_IMEMBERSHIP_CONDITION*

inherit

      ISECURITY_ENCODABLE (*3*)
          undefine(*1*)
             equals,
             to_string
          end

      ISECURITY_POLICY_ENCODABLE (*3*)
          undefine(*1*)
             equals,
             to_string
          end

      SYSTEM_OBJECT
          undefine (*2*)
             equals,
             to_string
          end

   feature -- Query

      equals (*a_obj*: *SYSTEM_OBJECT*): *BOOLEAN* **is** (*1*)
             -- dotnet_name: "IMembershipCondition.Equals (obj: Object):
             -- Boolean"
          deferred
          end

```
        to_string: SYSTEM_STRING is (1)
                    -- dotnet_name: "IMembershipCondition.ToString ():
                    -- String"
            deferred
            end
    …
    end
```

Table 34 Proxy class corresponding to the *.NET* interface *IMEMBERSHIP_CONDITION*

7. Generating Contract Wizard proxy classes

7.1 The *Eiffel* code generator

In this section, we have a closer look at the implementation of the *Eiffel* code generator (see Figure 19 below). The role of the *Eiffel* code generator is to generate the *Contract Wizard* proxy classes and to store them into the specified directory.
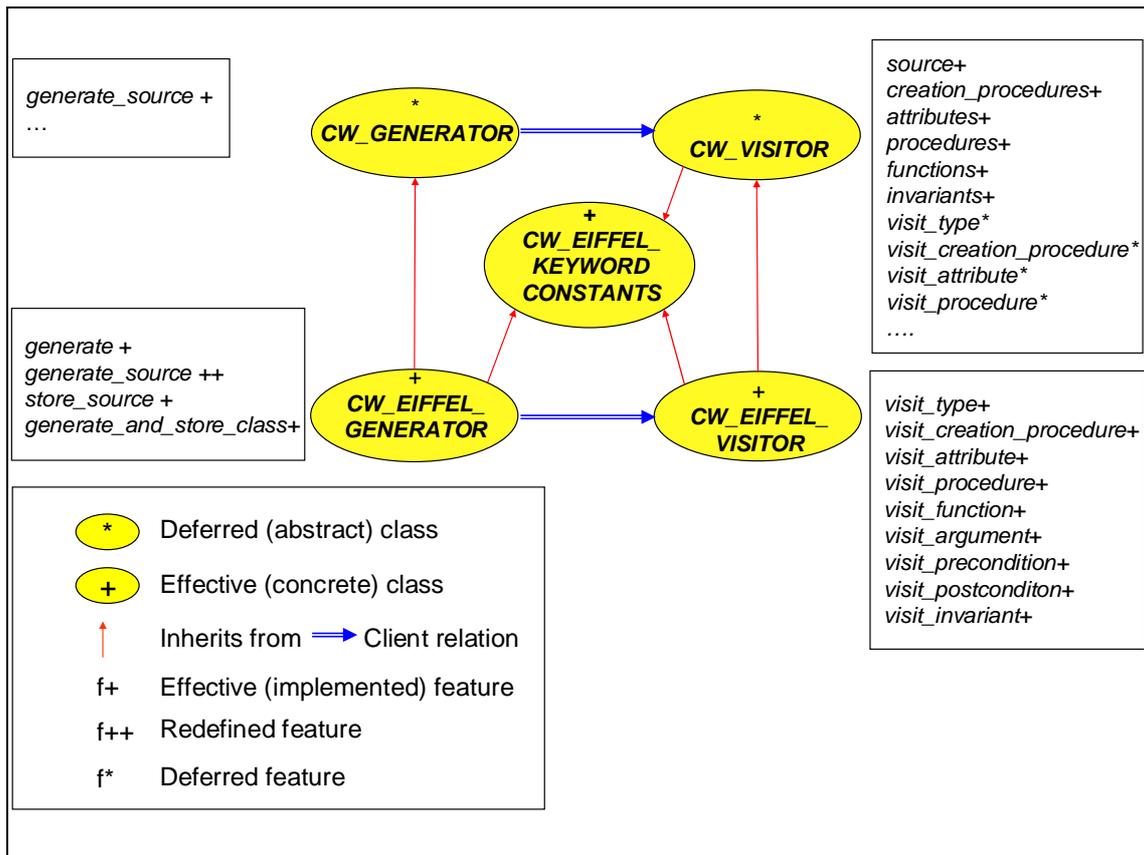
Figure 19: Class diagram of the *Eiffel* code generator

The *Eiffel* code generator consists of the deferred classes *CW_GENERATOR* and *CW_VISITOR* and of the effected classes *CW_EIFFEL_GENERATOR*, *CW_EIFFEL_VISITOR* and *CW_EIFFEL_CONSTANTS*.

CW_GENERATOR
The class *CW_GENERATOR* specifies the features common to all its descendants: *CW_EIFFEL_GENERATOR* (see *Figure 19*), *CW_XML_GENERATOR* and *CW_LACE_GENERATOR*. The class *CW_GENERATOR* and its descendants have a common creation procedure *make*. It has two arguments: *an_ast* (of type *LINKED_LIST* [*CW_TYPE*]) and *a_directory_path_name* (*STRING*). The argument *a_directory_path_name* defines the path to the directory where the generated code is stored. The data structure *ast* represents the parsed classes from the .*NET* assembly.

71

CW_VISITOR

The class *CW_VISITOR* defines features common to its descendants *CW_EIFFEL_VISITOR* and *CW_XML_VISITOR*. The last two classes are described below.

CW_EIFFEL_KEYWORD_CONSTANTS

The class *CW_EIFFEL_KEYWORD_CONSTANTS* specifies the required constants for generating the *Eiffel* representation of *Contract Wizard* proxy classes.

The *Eiffel visitor* (*CW_EIFFEL_VISITOR*)

The *Eiffel* code for the *Contract Wizard* proxy classes is generated using an *Eiffel visitor* (*CW_EIFFEL_VISITOR*). It specifies features for *Eiffel* code generation, namely:

- *visit_creation_procedure*: This feature generates *Eiffel* code for a *Contract Wizard* creation procedure (*CW_CREATION_PROCEDURE*) and appends it to the *creation_procedures* list. The code includes preconditions and postconditions.
- *visit_attribute*: This feature generates *Eiffel* code for a *Contract Wizard* attribute (*CW_ATTRIBUTE*) and appends it to the *attributes* list.
- *visit_procedure*: This feature generates *Eiffel* code for a *Contract Wizard* procedure (*CW_PROCEDURE*) and appends it to the *procedures* list. The code includes preconditions and postconditions.
- *visit_function*: This feature generates *Eiffel* code for a *Contract Wizard* function (*CW_FUNCTION*) and appends it to the *functions* list. The code includes preconditions and postconditions.
- *visit_invariant*: This feature generates *Eiffel* code for a *Contract Wizard* invariant (*CW_INVARIANT*) and appends it to the *invariants* list.
- *visit_type*: This feature generates *Eiffel* code for a *Contract Wizard* type (*CW_TYPE*) using the features *creation_procedures, attributes, procedures, functions* and *invariants*.

These features are called indirectly by registering the *Eiffel visitor* to a *Contract Wizard ast* node. A *visitor* is registered to a *Contract Wizard ast* node by the feature *visit*. A *visitor* can be registered more then once. There is an internal counter. You can ask every *Contract Wizard ast* node in which state it is. There are two states: *is_visited* and *not is_visited*. It can be very useful to distinguish between these two states. Some pre-processing can be done in the state *not is_visited*.

Example of how to use *visitors*

Table 35 on the next page shows an example how *Contract Wizard* proxy classes are generated using a *visitor*. Note that the *visitor* must be of type *CW_EIFFEL_VISITOR* to generate *Eiffel* code.

The *visitor* is registered twice to the *Contract Wizard* type (*CW_TYPE*) (1, 2) and once to every feature (*CW_FEATURE*) (3). After registering the *visitor* to all *Contract Wizard ast* nodes, the *Contract Wizard Eiffel* code can be retrieved using the query *visitor.type_text*.

```
visit_class (a_type: CW_TYPE) is
            -- Visit a_type.
    require
          type_not_void: a_type /= Void
          visitor_not_void: visitor /= Void
    local
          a_feature: CW_FEATURE
          features: LINKED_LIST [CW_FEATURE]
    do
          a_type.visit (visitor) (1)
          features := a_type.features
          from
                features.start
          until
                features.after
          loop
                a_feature := features.item
                visit_feature (a_feature) (3)
                features.forth
          end
          features := a_type.features
          visit_assertions (a_type.invariants)
          a_type.visit (2)
    end
```

Table 35: Registering a visitor to Contract Wizard ast nodes

Note: if you want to generate an *XML* representation of a *Contract Wizard* proxy class instead of the *Eiffel* representation, you must use the *XML visitor* (*CW_XML_VISITOR*) instead of the *Eiffel visitor* (*CW_EIFFEL_VISITOR*).

7.2 The *XML* code generator

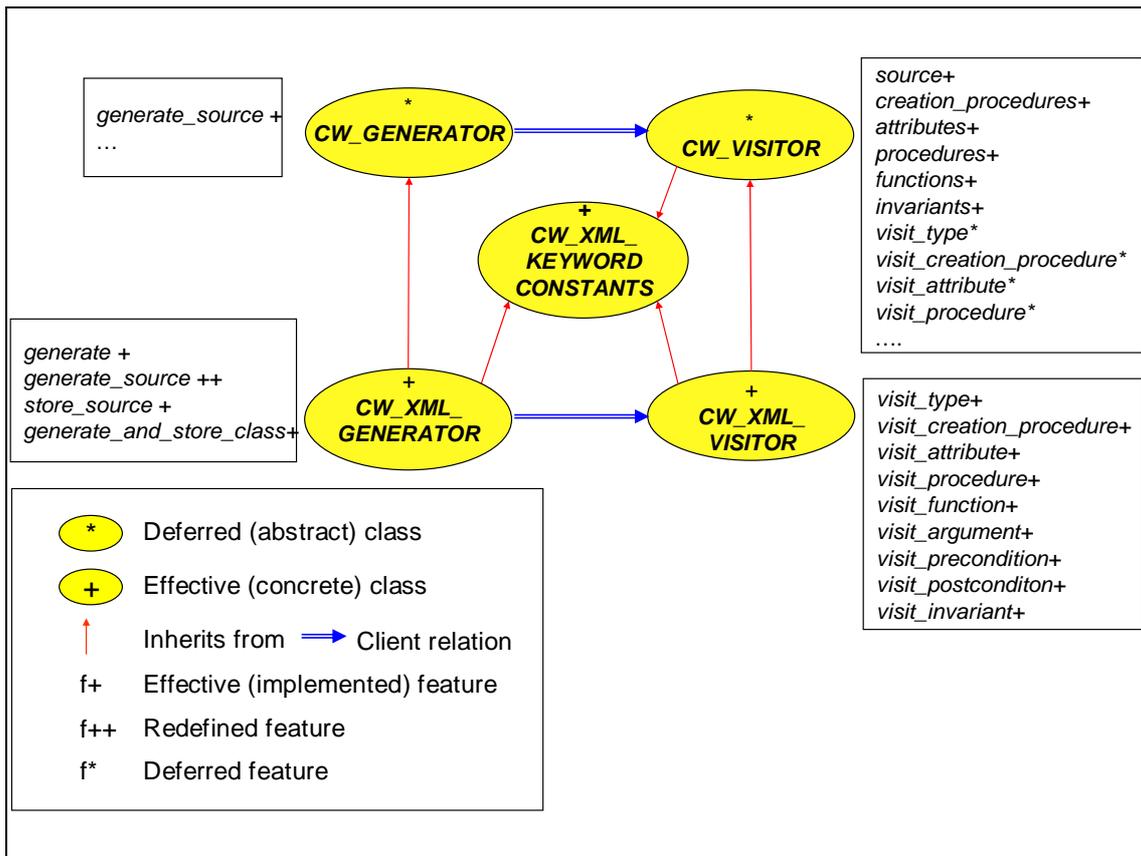The *XML* code generator resembles the *Eiffel* code generator in many ways.

Figure 20: Class diagram of the *XML* code generator

Figure 20 above shows how classes *CW_VISITOR*, *CW_XML_VISITOR CW_GENERATOR*, *CW_XML_GENERATOR* and *CW_XML_CONSTANTS* interact with each other.

The classes *CW_GENERATOR* and *CW_VISITOR* are the same as the ones discussed in section *The Eiffel code generator*.

CW_XML_GENERATOR

The class *CW_XML_GENERATOR* generates *XML* code using the *XML visitor* (*CW_XML_VISITOR*) and stores it to the specified *XML* file.

CW_XML_CONSTANTS

The class (*CW_XML_CONSTANTS*) specifies the *XML* tags required for generating *XML* proxy classes.

CW_XML_VISTOR

The generation of *XML* code is similar to the generation of *Eiffel* code, except that it uses an *XML visitor* instead of an *Eiffel visitor* in feature *visit_class* (see *Table 35*).

The class *CW_XML_VISITOR* is responsible for code generation. It has the following features:

- ***visit_creation_procedure***: Creates the *XML* representation of a *Contract Wizard* procedure (*CW_PROCEDURE*) including contracts. The generated code is appended to the *creation_procedures* list.

- ***visit_attribute*:** Creates the *XML* representation of a *Contract Wizard* attribute (*CW_ATTRIBUTE*) including contracts. The generated code is appended to the *attributes* list.
- ***visit_function***: Creates the *XML* representation of a *Contract Wizard* function (*CW_FUNCTION*) including contracts. The generated code is appended to the *functions* list.
- ***visit_procedure***: Creates the *XML* representation of a *Contract Wizard* procedure (*CW_PROCEDURE*) including contracts. The generated code is appended to the *procedures* list.
- ***visit_invariant*:** Creates the *XML* representation of a *Contract Wizard* invariant (*CW_INVARIANT*) including contracts. The generated code is appended to the *invariants* list.
- ***visit_type***: Creates the *XML* representation of a *Contract Wizard* type (*CW_TYPE*) making use of the features *creation_procedures, attributes, procedures, functions* and *invariants*.

You can access the generated XML code using the query *visitor.source*.

7.3 The *LACE* code generator

To compile the *Contract Wizard* proxy classes we need an *ACE* file. The *LACE* generator (*CW_LACE_GENERATOR*) (see below) creates the appropriate *ACE* file, written in *LACE* (Language for Assembly of Classes in Eiffel) and stores it into the specified directory.
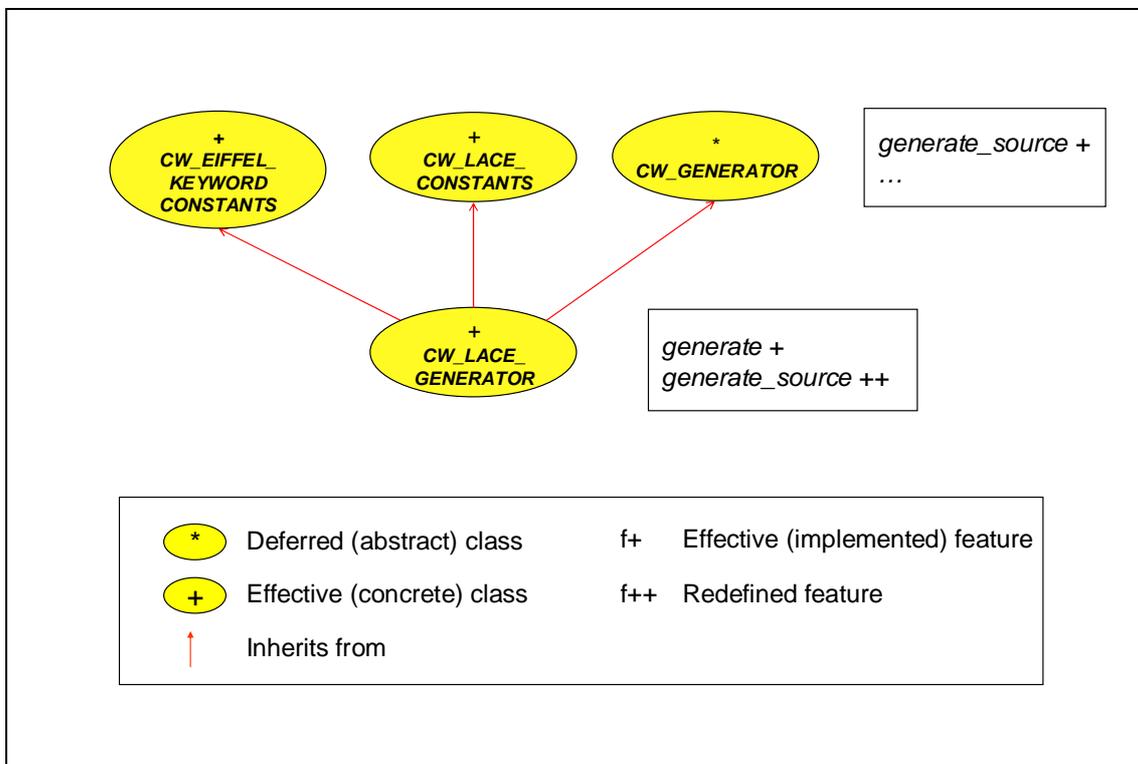

Figure 21: Class diagram of the *LACE* Generator

- ***CW_LACE_GENERATOR***
  The class *CW_LACE_GENERATOR* (see Figure 21 above) inherits from the three classes *CW_GENERATOR*, *CW_LACE_CONSTANTS* and *CW_EIFFEL_KEYWORD_ CONSTANTS*. The last two classes provide the *LACE* generator with the required constants to generate an *ACE* file.

- ***CW_GENERATOR***
  The feature *generate* (see Figure 21 above) provides the *LACE* generator with an assembly (*ASSEMBLY*, corresponding to the *.NET System.Reflection.Assembly*) and a *prefix*. The *prefix* is used to prevent ambiguous names.

- ***CW_LACE_GENERATOR***
  The *LACE* generator extracts all required information from the assembly to generate the *ACE* file, such as the assembly version, the assembly public key token and the assembly name.

8. Proxy class syntax

In this section, I discuss the syntax of the generated *Contract Wizard* proxy classes. I present the syntax on a small language based on regular expressions specified by the grammar defined in *Table 36* below.

Concatenation
A = B C: *A corresponds to B followed by C*

Option
A = B | C: *A corresponds to B or C*

Zero or one
A = [B]: *A corresponds to zero or one B*

Zero or many
A = {B}: *A corresponds to zero or many B*

Table 36: A small language based on regular expressions

I start with some basic concepts such as identifiers, formal and actual arguments, and types. Based on them I describe the syntax of creation procedures, attributes, procedures, functions and finally I explain the syntax of a *Contract Wizard* proxy class.

8.1 Identifier specification

Table 37 below shows the specification of an identifier (*Id*). Valid identifiers are for example: "a_name", "foo" etc.

Id = Letter { *Letter* | *Digit* | *Special_character* }

Letter = "A" | "B" | … | "Z" | "a" | "b" | … | "z"

Digit = "0" | "1" | "2" | … | "9"

Special_character = "_" | … | "*"

Table 37: Definition of identifiers

Note *Special_characters* are limited to characters provided by the .*NET* framework.

8.2 Type specification

Table 38 below shows the production of a type (*Type*). There are four different types handled by *Contract Wizard*:

- *Defined types*: Some types already exist in *Eiffel*. For these types *Contract Wizard* prepends the prefix *SYSTEM_*.
- *Array types*: These types represent arrays. The array type corresponds to the characters between the opening and the closing square bracket.
- *Object types*: These types represent objects.
- *Primitive types*: These are *CLS* compliant primitive types. They are not the same in *Eiffel* and in *.NET* (they have different names).

Defined_type = *"SYSTEM_STRING" | "SYSTEM_ARRAY" |*
"SYSTEM_CONSOLE" | "SYSTEM_DATE_TIME" | "SYSTEM_DIRECTORY" |
"SYSTEM_FILE" | "SYSTEM_QUEUE" | "SYSTEM_RANDOM" |
"SYSTEM_SORTED_LIST" | "SYSTEM_STACK" | "SYSTEM_STREAM"

Array_type = *"NATIVE_ARRAY" "["* *Object_type* | *Primitive_type* | *Defined_type*
"]"

Object_type = *Id*

Primitive_type := *"BOOLEAN" | "CHARACTER" | "INTEGER_8" |*
"INTEGER_16" | "INTEGER" | "INTEGER_64" | "POINTER" | "REAL"

Type = *Object_type* | *Primitive_type* | *Array_type* | *Defined_type*

Table 38: Definition of types

Valid productions for Type are: "*INTEGER_64*", "*NATIVE_ARRAY* [*INTEGER*]", "*MY_CLASS*" …

8.3 Formal argument specification

To express *Eiffel* routines, we have to introduce formal arguments. *Table 39* specifies a formal argument list.

Formal_argument = *Id* "*:*" *Type*

Formal_arguments = *Formal_argument* | [*Formal_argument* "*,*"
Formal_arguments]

Formal_argument_list = *"("* *Formal_arguments* "*)*"

Table 39: Definition of a formal argument

A valid production for a formal argument list (*Formal_argument_list*) is (*a_name*: *STRING*; *an_object*: *SYSTEM_OBJECT*).

8.4 Actual argument specification

Actual arguments are required for expressing *Eiffel* routine calls. Table 40 below specifies an actual argument list.

Actual_argument = *Id*

Actual_arguments = *Actual_argument* | [*Actual_argument* *","*
Actual arguments]

Actual_argument_list = *"("* *Actual_arguments* *")"*

Table 40: Definition of actual arguments

A valid production for an actual argument list (*Actual_argument_list*) is: (*a_name, an_object*).

8.5 Creation procedure specification

A *Contract Wizard* proxy class creation procedure consists of the keyword *frozen*, followed by the creation procedure name *make*, an optional formal argument list and a procedure body where the referenced *.NET* class is instantiated.
The restriction *frozen* is required to prohibit programmers from redefining the creation procedure.

Table 41 shows the production of a *Contract Wizard* proxy class creation procedure. *Reference* represents the references to the original *.NET* class.

Reference = *Id*

Creation_procedure = " frozen " " make " [*Formal_argument_list*] " is " " do "
"create " *Reference* "." "make " [Actual_argument_list] " end "

Table 41: Definition of a Contract Wizard creation procedure

Table 42 on next page shows a production of a valid creation procedure.

```
        frozen make (a_sum: INTEGER) is
            do
                create deposit.make (a_sum)
            end
```

Table 42: Example of a Contract Wizard creation procedure

8.6 Attribute specification

A *Contract Wizard* proxy class attribute represents a .*NET* field.

The *Contract Wizard* attribute has a *frozen* keyword in front of the attribute name prohibiting users from redefining it.

Table 43 below specifies a *Contract Wizard* proxy class attribute. *Reference* references the original .*NET* class.

```
Attribute_name = Id

Attribute = " frozen " Attribute_name ":" Type " is " " do " " Result" " := "
Reference "." Attribute_name "end"
```

Table 43: Definition of a Contract Wizard attribute

Table 44 shows a valid production of a *Contract Wizard* proxy class attribute.

```
        frozen balance: INTEGER is
            do
                Result := account.balance
            end
```

Table 44: Example of a Contract Wizard attribute

8.7 Procedure specification

A *Contract Wizard* proxy class procedure represents either a deferred (abstract) .*NET* procedure or an effected (implemented) .*NET* procedure.
If the *Contract Wizard* proxy class procedure represents a deferred .*NET* procedure, it consists of a procedure name, followed optionally by a formal argument list and a deferred procedure body.

If the *Contract Wizard* proxy class procedure represents an effected .*NET* procedure, there is a frozen keyword in front of the procedure name prohibiting users from redefining it. It has an effected feature body where the procedure call is delegated to the .*NET* procedure.

Table 45 below specifies a *Contract Wizard* proxy class procedure.

Procedure_name = *Id*

Deferred_procedure = *Procedure_name* [*Formal_argument_list*] " is "
" deferred " " end "

Effected_procedure = " frozen " *Procedure_name* [*Formal_argument_list*]
"is" "do" Reference "." *Procedure_name* [*Actual_argument_list*] " end "

Procedure = *Effected_procedure* | *Deferred_procedure*

Table 45: Definition of a Contract Wizard procedure

The two tables (*Table 46* and *Table 47*) below show examples of *Contract Wizard* proxy class procedures. The first table depicts a procedure representing an abstract .*NET* procedure. The second table shows a procedure representing an effected .*NET* procedure.

deposit (*a_sum*: *INTEGER*) **is**
    deferred
    end

Table 46: Example of a deferred Contract Wizard procedure

frozen *deposit* (*a_sum*: *INTEGER*) **is**
    do
        account.deposit (*a_sum*)
    end

Table 47: Example of an effected Contract Wizard procedure

8.8 Function specification

A *Contract Wizard* proxy class function represents either a deferred (abstract) .*NET* function or an effected (implemented) .*NET* function.

If the *Contract Wizard* proxy class function represents an abstract .*NET* function, it consists of a function name, followed optionally by a formal argument list, followed by a colon, a return type and a deferred function body. The function name represents the name of the .*NET* function. The function type represents the type of the .*NET* function.

If the *Contract Wizard* proxy class function represents an effected *.NET* function there is a *frozen* keyword in front of the function name prohibiting users from redefining it.
The delegated function call is assigned to the *Result* in the effected feature body.

Table 48 below specifies a *Contract Wizard* proxy class function.

Function_name = *Id*

Deferred_function = *Function_name* [*Formal_argument_list*] " : " *Type* " is " " deferred " " end "

Effected_function " := " " frozen " *Function_name* [*Formal_argument_list*] ":" *Type* " is " " do " " Result " ":= " *Reference* "." *Function_name* [*Actual_argument_list*] " end "

Function = *Effected_function* | *Deferred_function*

Table 48: Definition of a Contract Wizard function

The two tables (*Table 49* and *Table 50*) below show valid productions of *Contract Wizard* proxy class functions. The first table depicts a function representing an effected *.NET* function. The second table shows a function representing an abstract *.NET* function.

frozen add (*a_new_withdrawal*: SYSTEM_OBJECT): INTEGER **is**
    do
        Result := *withdrawal_list.add* (*a_new_withdrawal*)
    end

Table 49: Example of an effected Contract Wizard function

add (*a_new_withdrawal*: SYSTEM_OBJECT): INTEGER **is**
    deferred
    end

Table 50: Example of a deferred Contract Wizard function

8.9 Feature specification

Before we introduce *Contract Wizard* types, we have to specify features (*Features*) (see *Table 51* below). A feature (*Feature*) is a creation procedure, an attribute, a procedure or a function. For better readability we don't specify feature clauses.
Table 51 on next page shows the specifications of features.

Features = *Creation_procedure* | *Attribute* | *Procedure* | *Function*

Features = *Feature* | [*Features*]

Table 51: Definition of a Contract Wizard feature

8.10 Class specification

Now we are ready to specify *Contract Wizard* proxy classes. A *Contract Wizard* proxy class can act as a surrogate for the following *.NET* constructs:
- effected *.NET* classes
- deferred (abstract) *.NET* classes
- interfaces

The *Contract Wizard* proxy class inherits from the same interfaces as its corresponding *.NET* class. Every *Contract Wizard* proxy class inherits from *SYSTEM_OBJECT*.

The *Contract Wizard* Proxy class represents an effected *.NET* class
If the *Contract Wizard* proxy class represents an effected class, it has no *Type_distinctor* (see *Table 52* below) in front of the keyword *class*. It has a creation clause (*Creation_clause*) where it defines all creation procedures.

The *Contract Wizard* proxy class represents a deferred (abstract) *.NET* class
If the *Contract Wizard* proxy class represents a deferred class, it has the *Type_distinctor* *deferred* in front of the keyword *class*. The *Reference* in a deferred *Contract Wizard* proxy class points to a *deferred .NET* class. That is why it is not instantiated. To use a deferred *Contract Wizard* proxy class you have to effect it.

The *Contract Wizard* proxy class represents an interface
If the *Contract Wizard* proxy class represents an interface, it has the *Type_distinctor deferred* in front of the keyword *class*. It has no creation clause (*Creation_clause*) and no *Reference*.

Table 52 below specifies the syntax of a *Contract Wizard* proxy class.

Type_distinctor = [*" deferred "*]

Dotnet_class_type = *Id*

Class_name = *"CW_" Dotnet_class_type*

Interface_name = *Id*

Creation_procedure_names = *Creation_procedure_name* | [
Creation_procedure_name *","* *Creation_procedure_names*]

Creation_clause = *"create"* [*Creation_procedure_names*]

Undefinitions = *" undefine "* [*" get_hash_code "*] [*","*] [*" equals "*] [*","*] [*" to_string "*] *"end"*

Redefinitions = *" redefine"* [*" get_hash_code "*] [*","*] [*" equals "*] [*","*] [*" to_string "*] *"end"*

Interface = *Interface_name Undefinitions*

Interfaces = *Interface* [*Interfaces*]

Class_header = [*Class_distinctor*] *" class " Class_name " inherit "* [*Interfaces*] *" SYSTEM_OBJECT "* [*Redefinitions | Undefinitions*] [*Creation_clause*]

Ref = *" frozen " Reference " : " Dotnet_class_type*

Class_body = *Features* *" [feature {NONE} --Implementation " Ref*]

Contract_Wizard_proxy_class = *Class_header Class_body* " end"

Table 52: Definition of a Contract Wizard proxy class

84

9. Contract Handler

The Contract Handler provides the ability to insert contracts to the *Symbol DS*. It can be easily accessed from the *GUI* or any other client.

The Contract Handler is implemented by the class *CW_CONTRACT_HANDLER* and offers the following basic features:

- *add_precondition*
  Insert a precondition to the *ast* given a type name, a feature name, a tag and an expression representing the precondition.
- *add_postcondition*
  Insert a postcondition to the *ast* given a type name, a feature name, a tag and an expression representing the postcondition.
- *add_invariant*
  Insert an invariant to the *ast* given a type name, a tag and an expression representing the invariant.
- *type_by_name*
  Return the object representation of a type (*CW_TYPE*) given a type name.
- *feature_by_name*
 Return the object representation of a feature (*CW_FEATURE*) given a type name and a feature name.

Note: For future GUI developments, it would be better to access the data structure *ast* directly instead of using the Contract Handler (the data structure *ast* itself acts as a Contract Handler). Using the Contract Handler would yield a performance overhead.

10. Test cases

I tested *Contract Wizard II* on the *.NET* core library (*mscorlib.dll*) [14]. I checked that the generated *Contract Wizard* proxy classes are the same whether you retrieve them from *XML* files or from the .NET assembly.

Figure 22 below shows the results of performance tests with the *XML* parser and with the *.NET* parser. The left bar shows how long it takes to retrieve the *ast* from the *XML* representation of the *Contract Wizard* proxy classes representing *mscorlib*. The right bar shows how long it takes to retrieve the *ast* from the *.NET* assembly using the *.NET* reflection mechanism. As you can see, it is 2.4 times faster to retrieve the *ast* from the *XML* representation. Furthermore, the memory consumption to retrieve the *ast* from the *XML* representation is 30 percent lower (51 Megabyte (*XML*) compared to 78 Megabyte (.*NET*)).
I tested the parsing process on a system with a *Pentium* 3 processor with 1100 *MHz* and 256-megabyte RAM.
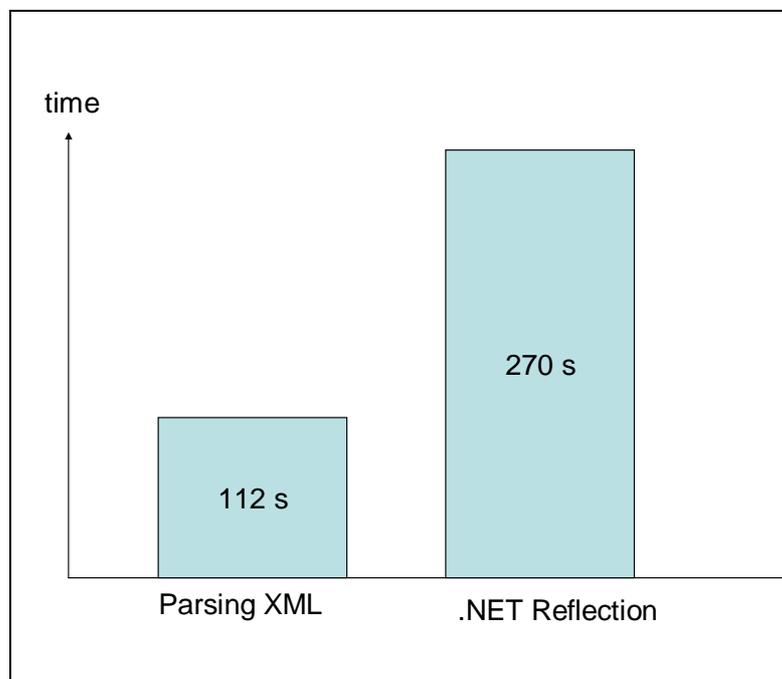
Figure 22: Performance test

I also tested how long it takes to add contracts using the *Contract Handler*.
Contract Wizard II requires in average less than 1 minute to add 5000 contracts (preconditions, postconditions and invariants) to *mscorlib* on the same system. We have tested adding contracts on randomly selected classes and features of *mscorlib*.
In the worst case there are O ($n*m$) computations to add a precondition and O (n) computations to add an invariant using the *Contract Handler* to the *symbol DS ast,* where n stands for the number of classes and m for the number of features.

Part D Theoretical part

11. Storing the contracts

The current implementation of *Contract Wizard II* stores contracts in an *XML* file. This has some advantages but also some drawbacks. Let us review them now.

11.1 Storing contracts in an *XML* file

Advantages

- *XML* files can easily be manipulated manually. This makes it possible to change contracts directly in the *XML* files.
- *XML* files can be read by humans, which can help to debug the *Contract Wizard* proxy classes.
- *XML* files are more flexible against future changes of the .*NET* framework.
- *XML* files also allow having more than one version of contracts for an assembly. (This may also be dangerous though.)

Disadvantages

- Consistency: We have to keep track of two files: the assembly and the *XML* representation.

11.2 Storing contracts in the assembly metadata

Another solution would be to store contracts in the metadata associated with any .NET assembly. Let us discuss the pros and cons of such an approach.

Advantages

- We do not need to generate an *XML* file, everything is in the same file, which probably implies a gain in space. The *XML* file representing the *Contract Wizard* proxy classes of the .*NET* core library (*mscorlib.dll*) requires 4.5 megabyte of space (compared with *mscorlib.dll* that requires about 2 megabyte of space).
- Because information is stored in just one file, it makes the assembly self-describing. No problem with extra files somewhere on the disk.
- It also makes the process of adding contracts simpler for the programmer.

Disadvantages

- Maybe lower performance: tests with the current version of *Contract Wizard II* have shown that it is faster to retrieve the symbol data structure *ast* from the *XML* representation versus getting it from the assembly using the reflection mechanism of *.NET* (see also *chapter 10*). However, it would remain to be tested how fast it is to extract contracts from the metadata.

11.3　Conclusion

I chose to store the contracts in an *XML* file representing the *Contract Wizard* proxy classes. I think using *XML* makes *Contract Wizard* more flexible and extendible. For example, if you want to rewrite *Contract Wizard* II for other programming languages you can reuse the same *XML* files.

12. Conclusion

I developed a completely new architecture for the tool *Contract Wizard*, now called *Contract Wizard II*, including the following points:

- Design of the software architecture.
- Implementation of a *.NET* parser and the corresponding data structure *ast*.
- An *XML* parser and the corresponding *XML* data structure to store the *Contract Wizard* proxy classes containing contracts as well. The *XML* parser also acts as the Contract Reader.
- Contract handler for managing the contracts within the data structure *ast.*
- Code generator, which generates the *Contract Wizard* proxy classes, their *XML* representation and the *LACE* file from the data structure *ast*.

Contract Wizard II works well. I tested it on the *.NET* core library (*mscorlib*) [14].

Because of time limit, I didn't implement a GUI but a command line application, which provides simple access to the functionality of *Contract Wizard* (see also *Demo Application*).

Future work

A lot of interesting work can be done:

- Implement support for inner classes and overloaded features.
- Join automatic contract extraction [5] with *Contract Wizard II*. *Contract Wizard II* offers a simple interface for insertion of contracts as *comma-separated* values (see chapter 2).
- Extend *Contract Wizard II* as a web service.
- Implement a *GUI.*

13. Acronyms

Acronym	Description
.NET	*"The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet."* [11] *The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework."* [11]
ACE	Assembly of Classes in *Eiffel*
AST	Abstract syntax tree
CLR	Common Language Runtime [14]
CLS	Common Language Specification [14]
COM	Component Object Model
CTS	Common Type System [14]
DS	Data structure
DTD	Document Type Definition [9]
GAC	Global Assembly Cache [14]
GUI	Graphical User Interface
LACE	Language for the Assembly of Classes in Eiffel
OOSC2	Object Oriented Software Construction Second Edition, the book by Bertrand Meyer, is used as the primary programming and language reference for implementing the software of this project in *Eiffel*.
Symbol DS	Symbol Data structure
XML	eXtendible Markup Language [9]

Table 53: Acronyms

References

[1]   Karine Arnout: "Développement de la technologie Eiffel dans l'environnement .NET de Microsoft", *Rapport de stage Jeune Ingénieur*, ENST Bretagne, September 2001.

[2]   Karine Arnout, and Raphaël Simon. "The *.NET Contract Wizard*: Adding *Design by Contract* to languages other than Eiffel", IEEE Computer Society, *TOOLS 39*, Santa Barbara, USA - July 2001, p: 14-23.

[3]   Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

[4]   Nancy Kotary, Colleen Gorman, Ellie Volckhausen, David Futato: *.NET Framework Essentials*, O'reilly, 2002

[5]   Christof Marti, *Automatic Contract Extraction*: *Developing a CIL parser*, http://se.inf.ethz.ch/projects/christof_marti/, consulted in September 2003

[6]   Bertrand Meyer: *.NET Training Course,* Prentice Hall, 2001.

[7]   Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[8]   Bertrand Meyer. "The start of an Eiffel standard". In *Journal of Object Technology*, Vol. 1, No. 2, July-August 2002, 95-99. http://www.jot.fm/issues/issue_2002_07/column8.pdf

[9]   Acronymfinder.com: http://www.acronymfinder.com, consulted in June 2003

[10]   ECMA Standard, http://www.ecma-international.org, consulted in June 2003

[11]   Eiffel Software, http://www.eiffel.com, consulted in June 2003.

[12]   Gobo Eiffel, http://www.gobosoft.com/eiffel/gobo/, consulted in June 2003

[13]   The Expat XML Parser, http://expat.sourceforge.net, consulted in June 2003

[14]   The Microsoft *.NET* library API: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netstart/html/cpframeworkref_start.asp?frame=true/, consulted in June 2003.

[15]   XML Spy, http://www.xmlspy.com, consulted in June 2003

[16]   O'Reilly Network_ C# in a Nutshell_ Introducing C# and the *.NET* Framework, Part 2 [Apr. 22, 2002]; Online at: http://www.ondotnet.com/pub/a/dotnet/excerpt/csharpnut_1/index2.html, consulted 21.06.2003