



IT University of Copenhagen
June 2012

When your applications run amok, use...

AMOC: A Visual Studio Extension for Application Modeling with Contracts

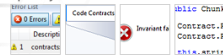
Alexandru F. Iosif-Lazăr
afla@itu.dk
270584-3263

Gabriel Balazs
gbal@itu.dk
201187-2843

Joseph R. Kiniry (supervisor)
josr@itu.dk



Specify code with code



Common Compiler Infrastructure
Code and AST Components

Domain-Specific Development
with Visual Studio DSL Tools



Abstract

Producing quality applications has always been a difficult challenge for software developers. Over the years, many solutions have been suggested for dealing with this issue such as evolved features for programming languages and various design and development methodologies.

Design by Contract is an approach for designing software based on formal specification of the program's features. It has been around for almost three decades and it has proven its worth in languages like Eiffel and language extensions like the Java Modeling Language. As an addition to forcing software designers and programmers to develop formal reasoning about their code, writing contracts also enables static design-time verification of programs. This allows the identification of possible errors before the application is even run.

Recently, Microsoft has shown interest in formal specification and verification for applications developed on the .NET Framework. Code Contracts enables writing contracts as method calls in .NET programming languages. Since using them is just writing regular C# or VB.NET code, Code Contracts have proven to be easy to learn but difficult to master. One of the downsides of Code Contracts is that it is hard to obtain an overview of the specifications just by looking at individual methods or classes in the code of large software applications.

We set out to demonstrate the benefits of enriching a class diagram with contracts, thus gathering both application structure and specification in one designer. After developing the main designer, we experimented with other features like:

- optimizing the layout of the model elements;
- using custom tool windows for actions that could not be done in the designer;
- writing shorthand forms of contracts to provide a compact view in the designer;
- pointing out contract inheritance throughout the model;
- contract optimization both on visual style and efficiency;
- generating design patterns with specifications based on existing model elements

The project was successful as we managed to include all these features in a single modeling tool called AMOC (Application MODELing with Contracts). The research and development process also led to new ideas for features that can further aid developers in understanding and making use of Code Contracts in a meaningful way.

Table of Contents

1.	Introduction.....	1
a.	Design by Contract.....	1
b.	Bridging Design by Contract and Model-Driven Architecture.....	2
c.	DbC in Visual Studio.....	3
d.	Goals	3
e.	Scope of paper: what we talk about and what we leave out.....	4
f.	Structure of the paper	4
2.	Methodology	6
3.	Current research in the area of MDA and DBC	7
a.	The 4-layer modeling architecture	7
b.	Eclipse toolkits	8
c.	EiffelStudio.....	10
d.	Code Contracts for .NET	11
4.	Tools	19
a.	Visual Studio Extensibility.....	19
b.	DSL Tools.....	21
c.	CCI - Common Compiler Infrastructure	25
d.	Project Roslyn	27
5.	AMOC features	28
a.	Visualization of code class architectures in VS2010.....	28
i.	Introduction	28
ii.	Class diagram in Visual Studio DSL Tools	28
iii.	Graphical elements in AMOC.....	31
iv.	Layout possibilities.....	32
v.	Customizing the layout of nested child shapes	33
vi.	Arranging shapes in the same NamespaceShape	36
b.	Model Visibility	37
c.	Contract editor.....	38
d.	Shorthand Contracts.....	42
e.	Contract inheritance	47
f.	Contract optimization.....	52
g.	Design Patterns with Contracts	58
i.	Double Derived	59
ii.	Abstract Factory	62

h.	Parsing code for building an architecture	67
i.	Various methods for parsing VS2010 solutions for files	67
ii.	Building the AST	67
iii.	Code-Model Comparison	74
6.	Discussion about goals in the project base	76
7.	Conclusions.....	79
8.	Perspectives.....	80
9.	Bibliography.....	81
10.	Appendices	84

1. Introduction

Ever since the creation of the first software applications, computer scientists have strived to make correct and fail-proof programs. A step was taken in this direction when the object oriented programming paradigm emerged. OOP enables code modularization—splitting code into classes, code reuse through inheritance, hiding code that is irrelevant to the client through encapsulation and a better way of handling similar functionality through polymorphism. All these tools facilitate the creation of software that is a lot easier to manage and to understand than code written in earlier languages. However bad code is still being written by developers: functions of hundreds of lines of code performing dozens of tasks in one call are one of the things that no one wants to inherit from former colleagues and yet, situations like these occur all too often.

Another step towards better software was taken when the first managed code runtime environments were launched. Since then, programmers do not have to worry about things like allocating and freeing memory or errors occurring due to bad memory management. Now there is someone else taking care of that. But there still are cases when, in the middle of using an application, we receive warnings about a null reference exception, or worse—we receive a generic error message like “Internal error”, “An error has occurred” or the worst—we do not receive any errors or warnings, but things just go really bad somewhere and we have no idea where to look. Catching and logging exceptions is a lifelong struggle for many.

The reason for poor exception handling is that programmers (and programs) do not take into consideration all possible input possibilities and execution paths. Instead, we often choose the quick and dirty solution that solves specific cases, leaving space for serious errors when unexpected input is given. It is not sufficient to validate user input. If we think about methods as individual functional units, unexpected input can be the values of the parameters that are passed when a method is called by other methods or the return values of other methods. Defensive programming promotes never trusting outside code and always writing validations for all possible inputs. However, in many cases the validation code gets mixed with the rest and makes programs difficult to understand and maintain.

a. Design by Contract

Design by Contract (DbC) is an approach to developing dependable software devised (and developed throughout the years) by Bertrand Meyer (1) (2) (3). Meyer also designed the Eiffel programming language which is entirely based on DbC.

In Meyer’s opinion the result of defensive programming “is simply to introduce more software, hence more sources of things that could go wrong at execution time, hence the need for more checks, and so on ad infinitum. Such blind and often redundant checking causes much of the complexity and unwieldiness that often characterizes software.” (3) Instead, he introduces the notion of contracts between pieces of code that interact (between a client and a supplier).

Contracts are conditions written as Boolean expressions: preconditions (that the client must ensure and the supplier can just assume as fulfilled), postconditions (that the supplier must ensure and the client can assume) and object invariants (conditions that are always met during the

lifetime of an object). Contracts provide multiple benefits. Contracts are a formal way of specifying program functionality: just by reading them one can understand what a method requires in order to execute and what it ensures upon execution. These specifications can be checked at runtime: if an error message is displayed when a contract fails at runtime, execution can be interrupted before the rest of the code can execute in a potentially erroneous manner. Contracts can be checked at design time: static checkers apply various forms of logical algorithms to prove if a contract might fail at runtime and suggest fixes to the developer.

Even though it has proven its usefulness in the Eiffel language, DbC is still not adopted on a large scale. Eiffel is mostly used for developing large financial or military software, but it is generally ignored for most domains. The Business Objects Notation has been created as an extension of Eiffel's modeling capabilities and describes both a methodology and a graphical model designer, but it is not officially implemented in the IDEs of any popular languages. "The only tool with BON support available over the years has been EiffelCase from Eiffel Software Inc, recently integrated into EiffelStudio"¹. For Java, DbC implementation is done through the Java Modeling Language (JML)², but to make use of it developers must learn a new language and how this language relates to Java.

Although for most developers it is not easy to adopt DbC methodologies, researchers are constantly striving both to improve DbC on a theoretical level (e.g., the Extended BON³) and create tools that make it easier to use (e.g., BONc⁴, Beetlz⁵).

b. Bridging Design by Contract and Model-Driven Architecture

A model is an abstraction of a thing, or as defined in the MDA Guide:

"A model of a system is a description or specification of that system and its environment for some certain purpose." (4) Users often build their problem model with the help of a graphical language such as the Unified Modeling Language (UML). A code generator or model compiler is able to generate some or all of the code from these models for the resulting application. Model Driven Architecture (MDA) is the initiative from the Object Management Group for this purpose: modeling and code generation. (5)

Thinking short-term in software engineering does not help adapting rigorous design principles as Design by Contract. Development managers are pressed by project schedules. According to the advocates of DbC, formal engineering disciplines save time when looking at the bigger picture as the software product will be of higher quality.

Using Model Driven Architecture and Design by Contract can provide time savings on the short term already, as invariants and pre/post-conditions can drive the code generators. Generating design patterns with built-in contracts also relieves the developer from defensive coding. Examples for tools that are parsing OCL and generate code from it accordingly are Sun's NetBeans

¹ http://www.bon-method.com/index_normal.htm

² <http://www.jmlspecs.org/>

³ <http://kindsoftware.com/documents/research/ircset/ebon.html>

⁴ <http://kindsoftware.com/products/opensource/BONc>

⁵ <http://kindsoftware.com/products/opensource/Beetlz>

Metadata Repository, the Adaptive Repository (6) and the Ecore modeling plugins for Eclipse. As Design by Contract is implemented in UML by the Object Constraint Language (OCL) (7), contracts in Ecore models are provided by OCL constraints to make the intent behind a model explicit (8).

c. DbC in Visual Studio

Visual Studio is the main IDE for Microsoft development (similar to Eclipse and NetBeans in the Java world), hosting rich tools for various popular programming languages (VB.NET, C#, C++, F#, Iron Python) for the .NET Framework. In addition to the code related tools (like editors, compilers and debuggers) Visual Studio also has support for software modeling and design.

The Visual Studio Modeling Project⁶ is a special type of project that allows users to create UML models and use them in conjunction with other application projects. The Visual Studio Modeling Project has, in our view, three shortcomings:

1. it is only available to Visual Studio Ultimate users;
2. it can only link model elements to Team Foundation Server work items;
3. it does not have full contracts support.

Among the properties of operations in UML Class Diagrams we find “postconditions” and “preconditions”, but these properties are simple lines of text that do not have any use besides telling human users how the operations should behave.

While the Modeling Project represents “Design without Contracts” another Microsoft project, Code Contracts, implements “Contracts without Design”. Code Contracts are implemented as methods in a .NET library. They support both runtime and static checking. However, writing contracts for large software applications can be difficult without the overview provided by modeling tools.

d. Goals

We seek to bridge the gap between the model and the code, between the design and the specifications by extending Visual Studio with our own tool. To fully implement DbC, the tool must be a model designer with the ability to include contracts that can be translated into Code Contracts. Additionally, we will try to implement static analysis of contracts in the context of the model and to find a visually compact and easy to use representation for both the model and contracts. We call this designer AMOC (Application MOdeling with Contracts).

We have identified a list of goals and “nice to have” features that guided us through the project.

Goals:

1. AMOC should help developers model their application in a usable way.
2. It should provide an easy transition from model to code and from code to model.
3. Display inconsistencies as errors or warnings upon saving a model.

⁶ <http://msdn.microsoft.com/en-us/library/dd409445>

4. It should make use of Visual Studio Extensibility such as Domain Specific Language Tools and Add-ins.
5. It should help .NET developers make use of .NET Code Contracts in a simplistic way.

“Nice to have” features:

6. Working in both synchronized and asynchronous modes (between model and code).
7. Research ways for creating a modeling tool, taking the concerns of the current DSL developers into consideration.
8. Representing domain specific languages in a visual and textual format
9. Visualization of a contract in a graphical invariant designer
10. Error messages stating the violation of a constraint in a friendly language (connecting contracts to user-understandable error messages).
11. Check contract expression style, undefinedness and perform other checks that the C# compiler and the Code Contracts Static Checker are not able to.
12. Automate the suggestion of contract templates for implementation of design patterns.

e. Scope of paper: what we talk about and what we leave out

In this paper we present the results of our project. We explain AMOC’s features and the research process that led to them. We describe the various Microsoft research projects that we have used to seamlessly integrate AMOC with the existing Visual Studio features and development style.

We do not attempt to change the DbC theory, but to apply it in a meaningful way in the .NET world. Also, we do not cover the basic features of Visual Studio or the C# programming language. Thus, we hope that this paper (and tool) will be useful to people that already have knowledge of Microsoft programming technologies and wish for an alternative to defensive programming.

f. Structure of the paper

Following the **Introduction** chapter, in chapter 2 we briefly mention what methodology we have used throughout the projects course.

Chapter 3 introduces our inspiration sources, listing some of the main tools that support Design by Contract. We also present Code Contracts, how they are used in .NET projects and how they work.

Chapter 4 describes the toolkits we experimented with, pursuing the idea of creating a modeling tool that implements Design by Contract based on the inspirations. We describe general Visual Studio Extensibility options and several Microsoft research projects: Domain-Specific Language Tools, the Common Compiler Infrastructure and Roslyn.

Chapter 5 lists AMOC’s features with insight to implementation details and theoretical background.

In chapter 6 we shortly discuss what we managed to achieve from our original set of goals and sets up basic grounds for further research.

Conclusions and further perspectives follow in chapter 7 while closing the paper by relating to research literature.

Snapshots of our initial ideas are presented in the Appendices to help the reader follow our train of thought.

2. Methodology

During the conception phase of the project we reviewed various modeling languages (e.g., UML + OCL, BON) and tools (the Visual Studio Modeling Project, Eiffel, and the Graphical Modeling Project for Eclipse). We have also studied research papers about Design by Contract, formal specifications and verification. From all these we made a list of features that we found interesting, that have been implemented in other tools and have proven successful or features that were never implemented, but seemed to have a great potential.

After forming a general idea of what we want to accomplish with this project we began researching Visual Studio Extensibility and other Microsoft technologies such as DSL Tools, Code Contracts and the C# programming language. We continuously updated the initial list of features removing those that would not integrate well in the Visual Studio IDE and adding new ones.

We also created drawn prototypes (see Appendix 3) that helped us envision the user interface and guide us through the creation of the first few features.

As we started to write code, we generally followed the principles of continuous integration:

- we set up an online SVN repository on Assembla⁷;
- we committed to the baseline as often as possible;
- we build the solution before every commit;
- we kept our local code up to date to make sure that the features we work on do not break the code in the repository

In the beginning of the project we had also envisioned a test and feedback phase. The idea was that once we developed an initial presentable version of AMOC, we would upload it on a public website to get feedback from online communities of Visual Studio users, to shape further development. Such communities would also provide an efficient way of testing the product (acceptance tests), improving its general quality. However, until the moment at which this report is written, we did not reach a version that we would proudly present to the public.

⁷ <http://www.assembla.com/>

3. Current research in the area of MDA and DBC

We found inspiration for our project in Eclipse extensions and EiffelStudio. This chapter is going to list these inspirational features and briefly introduce MDA.

a. The 4-layer modeling architecture

A domain is defined in “Generative Programming: Methods, Tools, and Applications” by Krzysztof Czarnecki as: “An area of knowledge scoped to maximize the satisfaction of the requirements of its stakeholders, including a set of concepts and terminology understood by practitioners in that area, and including knowledge of how to build software systems (or parts of software systems) in that area.” (9)

Domain-Specific Development has been a topic in software engineering since the “On the Design and Development of Program Families” paper written by David Parnas was published in 1976. It presented a possibility for program family generation. Another very influential book about design patterns called “Design Patterns: Elements of Reusable Object-Oriented Software” got published in 1994. This book was written by Gamma, Helm, Johnson, and Vlissides and currently is referenced in scientific software engineering papers as the GoF or “Gang of Four” book. The “Gang of Four” has described the Interpreter pattern, which has the following intent: “Given a language, define a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.”

However Domain-Specific Development only started to be generally accepted in software engineering and the IT industry in the recent years. Domain-Specific Development is related to the Model-Driven Development initiative. Now there are many model-driven software development tools that come from many different vendors. These development tools provide the users with the ability of building a model of their recurring problem. (5)

The M3 level is the meta-meta model level in the 4-layer architecture from OMG (Object Management Group)⁸. This level is reserved for all the generic modeling technologies, facilities and tools. (4)

The M2 meta-model level is for the conceptual or information model, where the important model concepts are described. Our basic UML and the full UML defined by MOF lie on this level. A general purpose programming language like C# or Java is also considered as an M2 model.

The M1 model level realizes the meta-model concepts by constructing corresponding relationships between the specialized problem domain objects. This model is called the data, logical or domain model. It includes all the data structures and classes together with their relationships. The logical model is the basis of use cases. Constraints and invariants are often incorporated on this level. An example is a C# class that contains Code Contracts statements on the methods (UML operations are called methods in C#).

The instance level (M0) is the physical model that realizes the logical model as a direct implementation in a target technology such as a running instance of a C# or Java object.

⁸ <http://www.omg.org/>

b. Eclipse toolkits

Eclipse is the official IDE of the Java general purpose programming language. The popular modeling toolkit extension EMF (Eclipse Modeling Framework)⁹ includes the Ecore meta-model for describing models in UML (Unified Modeling Language) fashion, EMF.Edit for building Eclipse editors for the EMF models and EMF.Codegen that uses the JDT (Java Development Tooling) component from Eclipse to generate code for the EMF model editors.

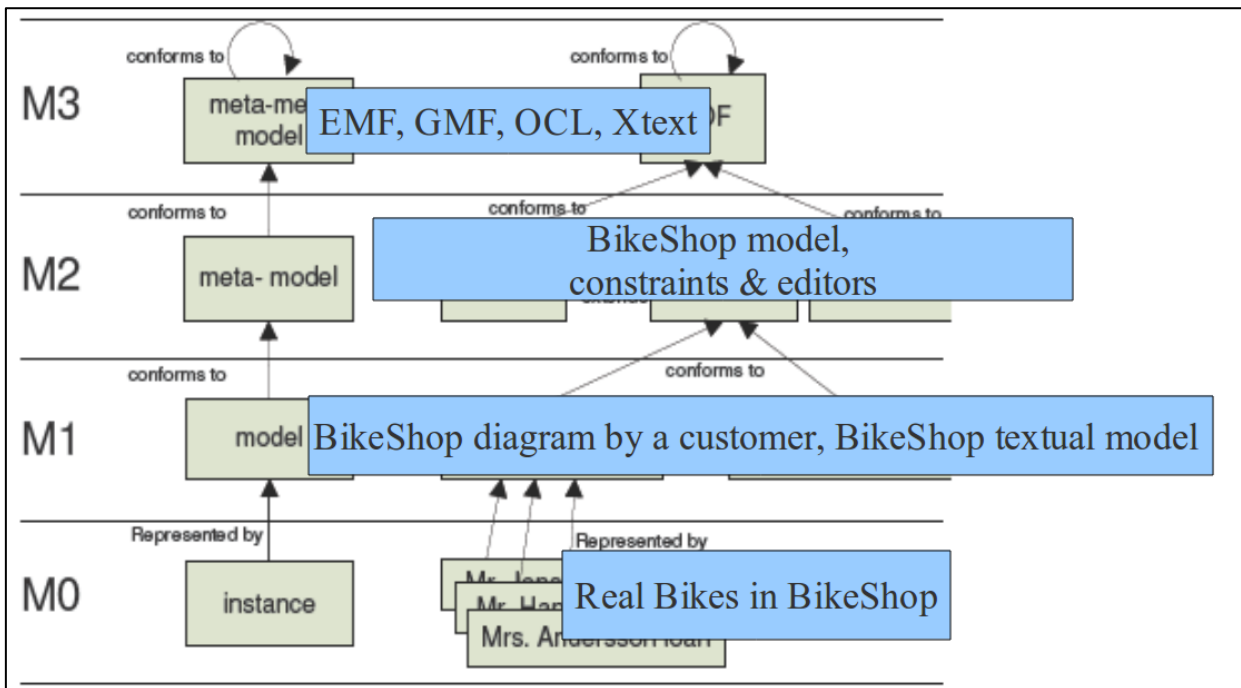


Figure 1: Modeling bikes in a BikeShop DSL using Ecore related toolkits and OCL (10)

We present the Eclipse Modeling Framework based on the BikeShop DSL. This DSL models a problem domain that is restricted to the domain of ordering custom bikes in a bike shop. On Figure 1 the OCL model as technology resides in the meta-metamodel (M3) level of OMGs modeling architecture. Its alternative in the .NET world is the Code Contracts API. The original illustration of Figure 1 can be found at the extremely influential book “Model-Driven Software Development: Technology, Engineering, Management” by Thomas Stahl, Markus Voelter and Krzysztof Czarnecki. (10) OCL constraints themselves are specifications on the three lower levels of the modeling architecture. These constraints specify the intent of models on M2, M1 and M0 levels.

The same M3 layer is reserved for MOF (Meta Object Facility) which defines UML and the tools using Ecore models that support defining DSLs in Eclipse: EMF for modeling the problem domain (creating a logical model), GMF (Eclipse Graphical Modeling Framework)¹⁰ for creating a graphical DSL for this model and Xtext for creating a textual DSL for the model.

⁹ <http://www.eclipse.org/modeling/emf/?project=emf>

¹⁰ <http://www.eclipse.org/modeling/gmp/>

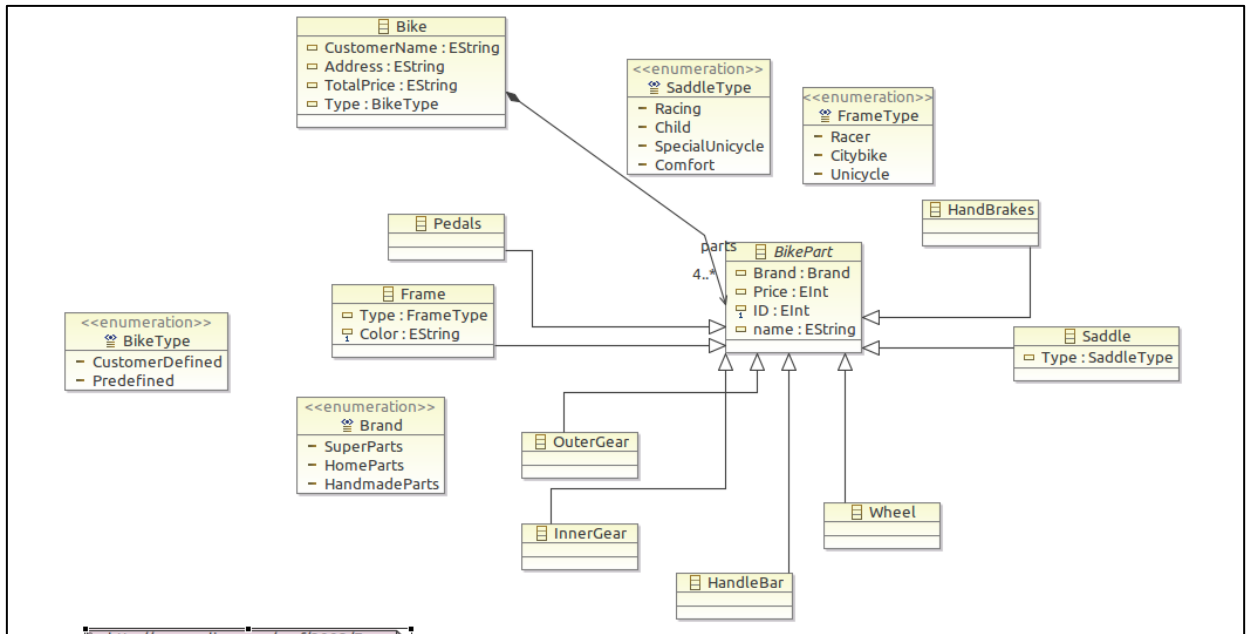


Figure 2: Metamodel (M2) of the BikeShop DSL, presented by the EMF modeling tool

Figure 2 shows the metamodel (M2) of BikeShop DSL, defined by Ecore model elements. Any custom bike consists of bike parts and has a customer that creates the custom bike using bike modeling tools in the bike shop. A multiplicity constraint on this model specifies that a bike must consist of minimum 4 bike parts in order to be valid. Multiplicity constraints can be combined with other constraints that reside in the background specification of this model as OCL constraints.

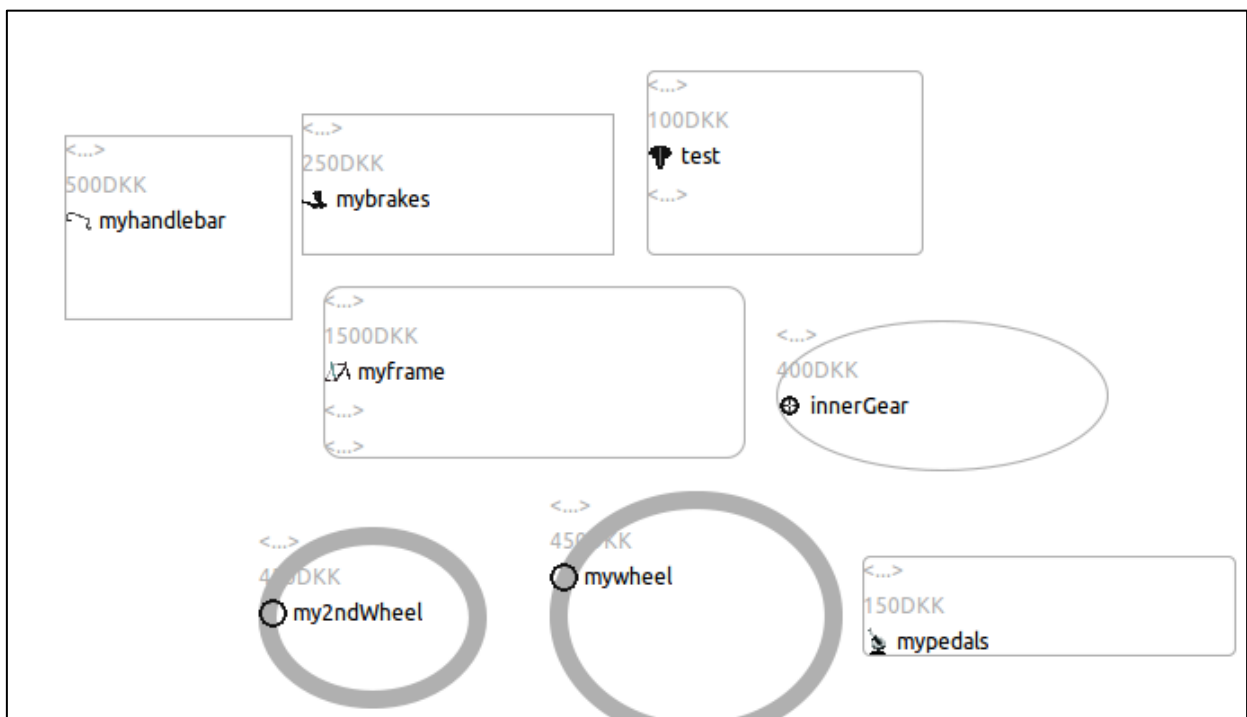


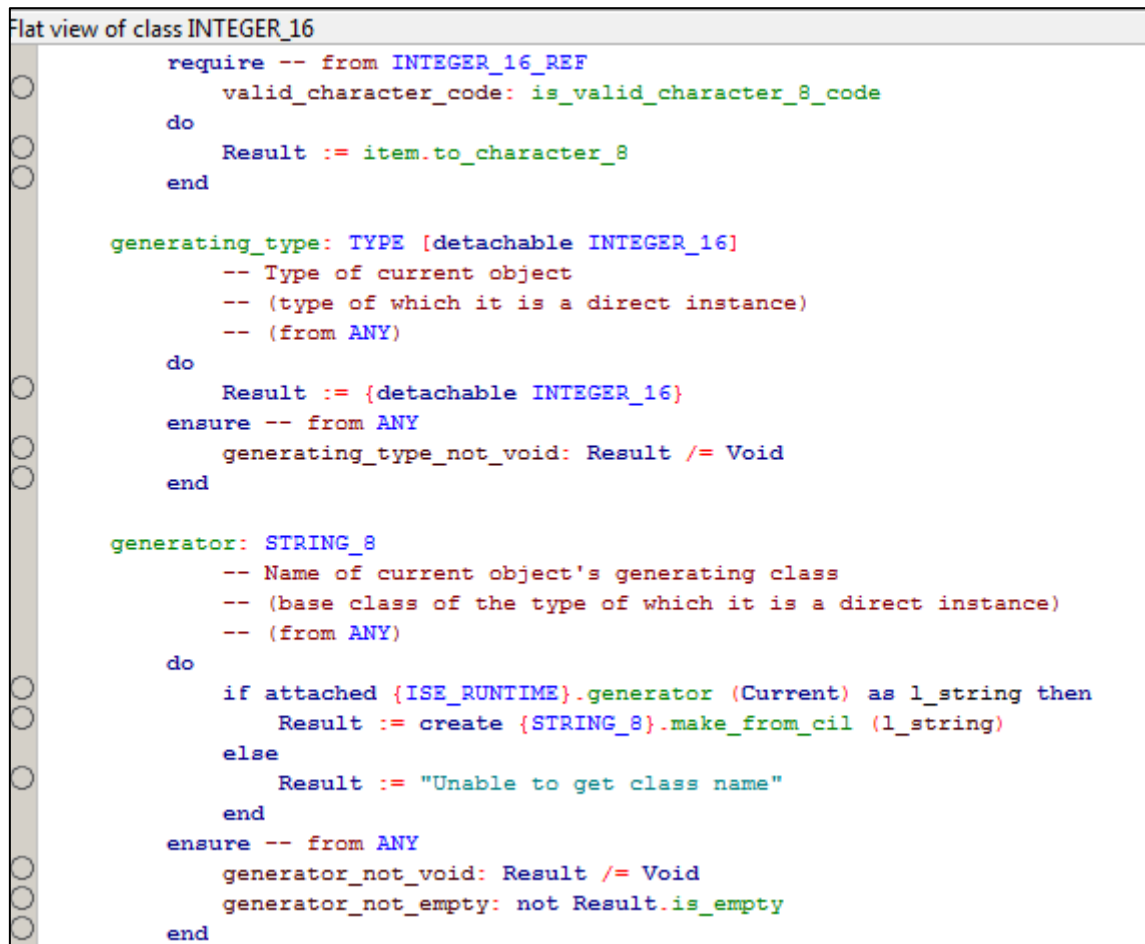
Figure 3: Model (M1) of a citybike "system" created and presented by the BikeShop DSL

Figure 3 presents some of the possibilities using GMF for creating a domain specific model. On this figure, the customer models a bike that has two wheels, a pair of pedals, brakes, a saddle, a handlebar and an inner gear. The visuals of shapes are very customizable. The OCL constraints specified for the DSL are inherited to the graphical diagram and the textual syntax, but are not visible on the models, only as a list of specifications in the background. It is also possible to specify when each constraint should be validated. Constraints can prevent creating invalid models in both visual and textual models, or on the users request to validate the model.

In this bike shop domain, an MO model would be the order sent to the suppliers of the bike shop and the concrete bike delivered to the customer.

c. EiffelStudio

EiffelStudio has vast modeling capabilities already built-in, as well as support for the Design By Contract principle. EiffelStudio is the official IDE for the Eiffel general purpose programming language - the language where the Design by Contract idea was first implemented (8). Below is the description of the features that became the inspiration for some of AMOC's own features.



```

Flat view of class INTEGER_16

require -- from INTEGER_16_REF
    valid_character_code: is_valid_character_8_code
do
    Result := item.to_character_8
end

generating_type: TYPE [detachable INTEGER_16]
    -- Type of current object
    -- (type of which it is a direct instance)
    -- (from ANY)
do
    Result := {detachable INTEGER_16}
ensure -- from ANY
    generating_type_not_void: Result /= Void
end

generator: STRING_8
    -- Name of current object's generating class
    -- (base class of the type of which it is a direct instance)
    -- (from ANY)
do
    if attached {ISE_RUNTIME}.generator (Current) as l_string then
        Result := create {STRING_8}.make_from_cil (l_string)
    else
        Result := "Unable to get class name"
    end
ensure -- from ANY
    generator_not_void: Result /= Void
    generator_not_empty: not Result.is_empty
end
  
```

Figure 4: Flat View of *INTEGER_16*

A Flat View (Figure 4) shows a class with its own contracts and inherited contracts (11). This figure shows “require” and “ensure” keywords for contract preconditions and postconditions.

Section 5.d describes how we use these keywords as a replacement for **Contract.Requires** and **Contract.Ensures** from .NET Code Contracts. AMOC also features contract inheritance, as described in detail in section 5.e.

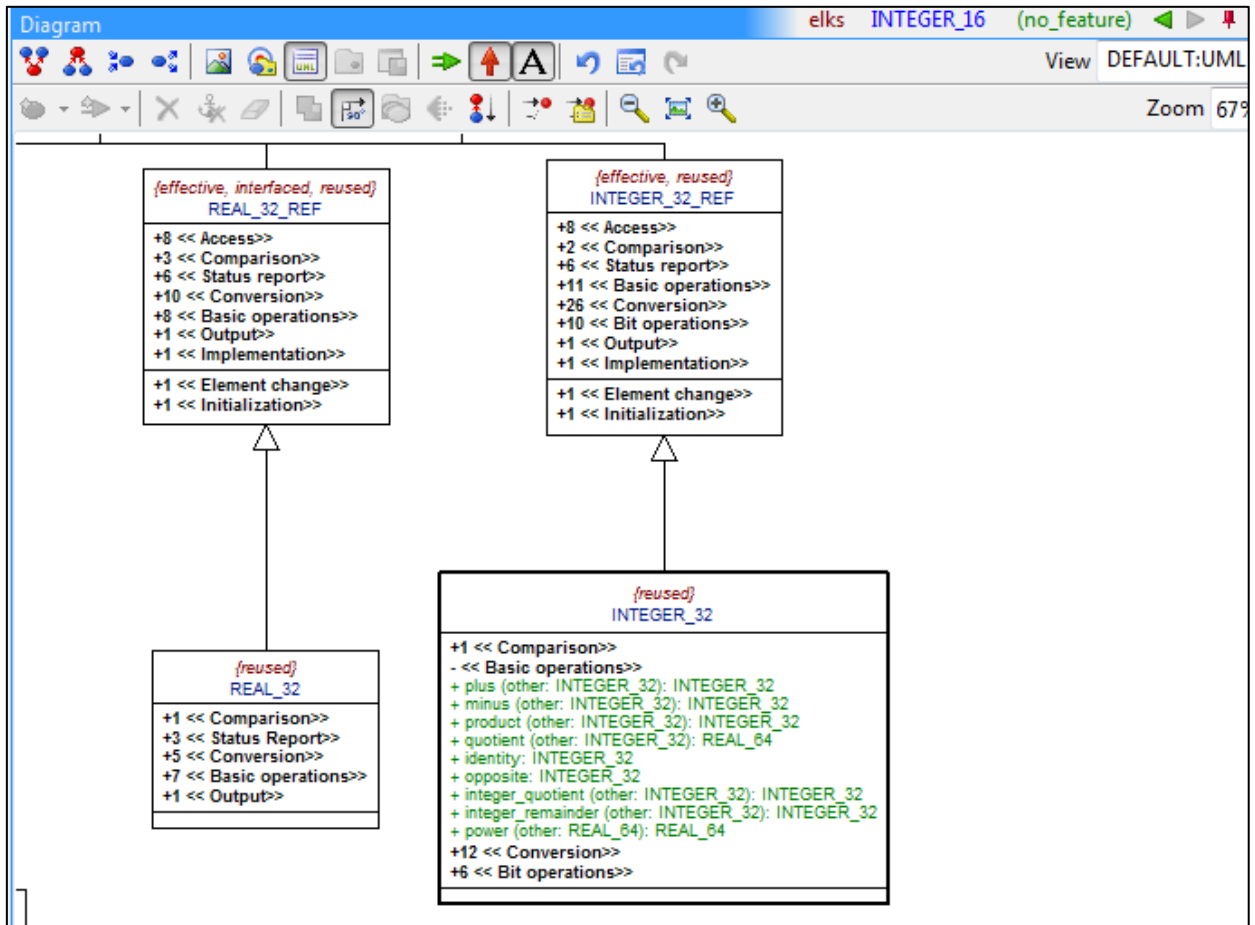


Figure 5: Diagram of *INTEGER_16* and its ancestors in UML. *Basic operations* expanded.

The Diagram tool in EiffelStudio is a mature product with numerous features. It shows a model with the help of UML or BON (Business Object Notation) general purpose modeling languages. However, it does not show the contracts on the model elements. Our solution was greatly inspired by the UML view in particular (Figure 5). The following features are used:

1. shape of the ModelElements;
2. border of ModelElements;
3. expanding the ModelElement for showing detailed information about it - as CompartmentShapes for operations.

d. Code Contracts for .NET

Code Contracts for .NET is a project developed by the Research in Software Engineering (RISE) group and Microsoft Research. It is an attempt to introduce the Design by Contract (DbC) paradigm to .NET development. The approach employed by Code Contracts allows the use of regular C# / VB.NET code to write specifications, eliminating the need to learn a new language.

Developers have the option to check the contracts both statically and at runtime. Additionally, by installing the Visual Studio 2010 Editor Extensions they benefit from:

- tool tip augmentation on method calls;
- inheritance adornments showing inherited contracts on method definitions;
- adornments on metadata files showing contracts for classes in other assemblies.

Embedded Contracts

While it builds on earlier Microsoft research (i.e. the Spec# programming methodology) (12), this project is using a different approach based on embedding contract specifications in a programming language (13).

Being a superset of C#, Spec# is adding new programming constructs for specifying pre- and postconditions or type support for differentiating between non-null and possibly-null references (14). On the other hand, Code Contracts are defined within the C# programming language. Writing a pre- or postcondition is equivalent to calling the right method in the right place. Specification annotations such as marking a method as pure are regular C# attributes.

```
1. Contract.Requires( x != null );
2. Contract.Requires<ArgumentNullException>( x != null, "x" );
3.
4. if ( x == null ) throw new ...
5. Contract.EndContractBlock(); // All previous ' if ' checks are preconditions
6.
7. Contract.Ensures( this .F > 0 );
8. Contract.EnsuresOnThrow<T>( this.F > 0 );
9.
10. Contract.Ensures(0 < Contract.Result<int>());
11. Contract.Ensures(Contract.Result<int>() < Contract.OldValue(x));
12. Contract.Ensures(Contract.ValueAtReturn(out x) == 3);
13.
14. [ContractInvariantMethod]
15. private void ObjectInvariant () {
16.     Contract.Invariant( this .y >= 0 );
17.     Contract.Invariant( this .x > this.y );
18.     ...
19. }
20.
21. Contract.ForAll (xs , x => x != null)
22. Contract.ForAll (0, xs.Length, index => xs[index] > 0));
23.
24. Contract.Exists (xs , x => x != null)
25. Contract.Exists (0, xs.Length, index => xs[index] > 0));
```

Listing 1: Contract methods

The Code Contracts User Manual (15) provides a detailed description of the contracts. All contracts are public, static methods of the **Contract** class in the **System.Diagnostics.Contracts** namespace (Listing 1).

Preconditions are specified by calling one of the overloads of the **Contract.Requires** method. The example in Listing 1 line 1 is the most basic precondition method, taking as parameter the Boolean expression to be checked. The example on line 2 takes as a generic parameter the type of exception to be thrown in case the precondition fails. Another kind of precondition is the “legacy requires”: one or more if-then-throw constructs followed by the **EndContractBlock()** method (lines 4 and 5).

Postconditions are specified by using the **Contract.Ensures** method. The overload on line 7 specifies a condition that must be checked if the method execution ends without throwing an exception. The overload on line 8 takes as a generic parameter an exception type; the condition will be checked when an exception of that type bubbles up from the method.

Postconditions make use of special methods to access values that exist at certain moments during the execution. Placing a postcondition on the result returned by a method is done using the **Result<T>()** method (line 10). Since at runtime the postconditions are actually checked right before the method execution ends, one might need to reference the initial value of a parameter. This is done using the method **OldValue<T>(T e)** (line 11). If a parameter has the **out** modifier and a postcondition must be placed on its value on method exit, the **ValueAtReturn<T>(out T e)** method can be used (line 12). There are several limits to these methods: they can only be used in postconditions and have specific interaction behavior with each other and with quantifier methods.

Object invariants must be specified within an instance method marked with the **ContractInvariantMethod** attribute. The contracts themselves are calls to the **Invariant(bool condition)** method. “Object invariants are conditions that should hold on each instance of a class whenever that object is visible to a client”. Their runtime behavior is best explained in the Code Contracts User Manual: “During runtime checking, invariants are checked at the end of each public method. If an invariant mentions a public method in the same class, then the invariant check that would normally happen at the end of that public method is disabled and checked only at the end of the outermost method call to that class.” (15)

Quantifier methods are helper methods that enable checking a condition over the elements of a collection or within an integer interval (lines 21 to 25). They can be used in any contract methods (pre-, postconditions and invariants), but their interaction with other special methods is restricted.

Additional contract methods are the **Assert(bool condition)** and **Assume(bool condition)**. At runtime, they behave the same way: if the condition is false, an exception is thrown. Upon static checking, the **Assume** condition is assumed true and added to the checker’s list of known facts, thus adding to the base of further logical processing.

Contract inheritance

Code Contracts inheritance is complying with the restrictions of behavioral subtyping (the Liskov Substitution Principle) as they are defined in “A behavioral notion of subtyping” by Barbara Liskov and Jeanette Wing (16):

1. preconditions cannot be strengthened in a subtype;
2. postconditions cannot be weakened in a subtype;

3. invariants of the supertype must be preserved in the subtype.

Since Code Contracts are inherited by methods from implemented interface methods or overridden superclass methods these restrictions are met.

A possible consideration to follow these statements could be this: if preconditions cannot be strengthened then it should be possible to weaken them. Respectively, if postconditions cannot be weakened then it should be possible to strengthen them.

```

1. class Engine
2.   string[] parts
3.   int Insert (IDBContext dbContext)
4.     requires dbContext.SupportsTransactions
5.     // code for inserting engine and parts
6.     // using the context's transaction manager
7.     ensures result > 0
8. end
9.
10. class MyEngine : Engine
11.   int transactionCount = 0
12.   override int Insert (IDBContext dbContext)
13.     // code for inserting engine and parts using a custom transaction manager
14.     ensures transactionCount == transactionCount' + 1
15. end

```

Listing 2 Liskov substitution example

Consider the pseudocode example in Listing 2. An entity called **Engine** is exposing an **Insert** operation which takes as parameter an object that implements a database context. The operation depends on the context's transaction support so the precondition on line 4 checks for it. The operation also defines a postcondition to ensure that the return value is greater than zero.

MyEngine is inheriting the **parts** property from **Engine**. The **MyEngine.Insert** operation is inheriting the contracts from **Engine.Insert**. However, **MyEngine.Insert** implements its own transaction mechanism. This is a case when the precondition should be weakened to allow even transactionless database contexts to be passed as parameters.

In the same time **MyEngine.Insert** strengthens the postcondition to ensure that, beside the return value being greater than zero, the **transactionCount** is incremented.

Code Contracts allow strengthening postconditions by adding **Contract.Ensures** methods, but lack a mechanism for weakening preconditions by specifying which should be inherited and which should not. Hence when a method implements or overrides another and the same time defines an additional precondition, a warning will be displayed.

"If a method has multiple root methods (overrides a method and at the same time implements an interface method (or implements two different interface methods)), then the effective precondition of the implementing method would be the conjunction of the preconditions of all root methods. Since this may be a stronger precondition than the precondition of any of the root

methods, this is not allowed [...] thus [the tools] emit a warning in situations where there are multiple root methods and at least one has a precondition.” (15).

The static checker (codename Clousot)

The Code Contracts static checker attempts to discover possible problems within the developed application before it is actually running. As it is stated in the publication “Static contract checking with Abstract Interpretation” by Manuel Fähndrich and Francesco Logozzo, Clousot “is based on abstract interpretation instead of solely relying on a theorem prover” (17). Abstract interpretation is used in ESC/Java (18) or Boogie for Spec# (19). Abstract interpretation presents several advantages, such as being more automatic than theorem provers (abstract interpretation can automatically compute loop invariants), but it also has disadvantages (sometimes fails to infer facts that seem trivial from a logical point of view and requires the insertion of explicit assumptions using the **Assume** method (20)).

One of the interesting aspects of Clousot is that the source on which the contracts are checked is not the C# or VB.NET code, but the compiled CIL (Common Intermediate Language) code. There do not need to be individual versions of Clousot for each .NET language. The checker cannot benefit from the abstraction of high-level languages and it must extract the contracts from CIL code to be able to properly report warnings and suggestions to the user.

Static and runtime checking of Code Contracts can be enabled or disabled independently. If runtime checking is disabled it means that the contracts are not compiled into the assembly. The following question arises: how is it that the contracts can be compiled to allow static checking, but are ignored at runtime?

```

1. public int Increment(int i)
2. {
3.     Contract.Requires(i > 0);
4.     Contract.Ensures(Contract.Result<int>() == i + 1);
5.     return i + 1;
6. }
7.
8. .method public hidebysig instance int32 Increment(int32 i) cil managed
9. {
10.    // Code size          9 (0x9)
11.    .maxstack 2
12.    .locals init ([0] int32 CS$1$0000)
13.    IL_0000: nop
14.    IL_0001: ldarg.1
15.    IL_0002: ldc.i4.1
16.    IL_0003: add
17.    IL_0004: stloc.0
18.    IL_0005: br.s      IL_0007
19.    IL_0007: ldloc.0
20.    IL_0008: ret
21. } // end of method Program::Increment

```

Listing 3: Compiled method in obj\Debug\assembly_name.exe

```

1. .method public hidebysig instance int32 Increment(int32 i) cil managed
2. {
3.     // Code size          28 (0x1c)
4.     .maxstack 8
5.     IL_0000: ldarg.1
6.     IL_0001: ldc.i4.0
7.     IL_0002: cgt
8.     IL_0004: call void
        [mscorlib]System.Diagnostics.Contracts.Contract::Requires(bool)
9.     IL_0009: call        !!0
        [mscorlib]System.Diagnostics.Contracts.Contract::Result<int32>()
10.    IL_000e: ldarg.1
11.    IL_000f: ldc.i4.1
12.    IL_0010: add
13.    IL_0011: ceq
14.    IL_0013: call        void
        [mscorlib]System.Diagnostics.Contracts.Contract::Ensures(bool)
15.    IL_0018: ldarg.1
16.    IL_0019: ldc.i4.1
17.    IL_001a: add
18.    IL_001b: ret
19. } // end of method Program::Increment

```

Listing 4: Compiled method in obj\Debug\Decl\assembly_name.exe

Let us consider the example in Listing 3. Lines 1 to 6 show a C# method with a precondition and a postcondition. Lines 8 to 21 show the compiled method; neither of the two conditions shows up. So where is the compiled code that Clousot is analyzing? When enabling static checking on a project, an additional assembly is built under the **obj** folder. This assembly is not copied to the **bin** folder so it cannot be referenced, but it can be used by the static checker. Listing 3 shows the same compiled method in this additional assembly.

Runtime checking

```

1. .method public hidebysig instance int32 Increment(int32 i) cil managed
2. {
3.     // Code size          77 (0x4d)
4.     .maxstack 4
5.     .locals init ([0] int32 CS$1$0000,
6.                  [1] int32 'Contract.Old(i)',
7.                  [2] int32 Contract.Result)
8.     IL_0000: ldarg.1
9.     IL_0001: ldc.i4.0
10.    IL_0002: cgt
11.    IL_0004: ldnull
12.    IL_0005: ldstr      "i > 0"
13.    IL_000a: call        void
        System.Diagnostics.Contracts.__ContractsRuntime::Requires(bool, string, string)
14.    IL_000f: nop
15.    .try
16.    {
17.        IL_0010: ldarg.1
18.        IL_0011: stloc.1
19.        IL_0012: leave     IL_0023
20.    } // end .try
21.    catch [mscorlib]System.Exception
22.    {
23.        IL_0017: brtrue     IL_001e
24.        IL_001c: rethrow
25.        IL_001e: leave     IL_0023
26.    } // end handler
27.    IL_0023: nop
28.    IL_0024: ldarg.1
29.    IL_0025: ldc.i4.1
30.    IL_0026: add
31.    IL_0027: stloc.0
32.    IL_0028: br           IL_002d
33.    IL_002d: ldloc.0
34.    IL_002e: stloc.2
35.    IL_002f: br           IL_0034
36.    IL_0034: ldloc.2
37.    IL_0035: ldloc.1
38.    IL_0036: ldc.i4.1
39.    IL_0037: add
40.    IL_0038: ceq
41.    IL_003a: ldnull
42.    IL_003b: ldstr      "Contract.Result<int>() == i + 1"
43.    IL_0040: call        void
        System.Diagnostics.Contracts.__ContractsRuntime::Ensures(bool, string, string)
44.    IL_0045: ldarg.0
45.    IL_0046: callvirt     instance void
        ConsoleApplication2.Program::$InvariantMethod$()
46.    IL_004b: ldloc.2
47.    IL_004c: ret
48. } // end of method Program::Increment

```

Listing 5 Compiled method in bin\Debug\assembly_name.exe

When runtime checking is enabled, the contracts are compiled into the final assembly under the **bin** folder. However, the original methods are removed and replaced by counterparts taking the original condition source code as string parameter. Listing 5 shows the same method as before, but this time it is compiled with contracts ready for runtime checking. This shows how local variables are initialized for initial parameter values and the method result. Also, the contract sources are passed as parameters to the **Requires** and **Ensures** methods. Another difference is that the two methods are called from the **__ContractsRuntime** class containing actual runtime behavior. And finally, at the end of this public method, the abstract invariant method is called.

The Contract Reference Assembly

```

1. .method public hidebysig instance int32 Increment(int32 i) cil managed
2. {
3.     // Code size          41 (0x29)
4.     .maxstack 9
5.     .locals init ([0] int32 V_0)
6.     IL_0000: ldarg.1
7.     IL_0001: ldc.i4.0
8.     IL_0002: cgt
9.     IL_0004: ldnull
10.    IL_0005: ldstr "i > 0"
11.    IL_000a: call void System.Diagnostics.Contracts.Contract::Requires(bool, string,
        string)
12.    IL_000f: call !!0 [mscorlib]System.Diagnostics.Contracts.
        Contract::Result<int32>()
13.    IL_0014: ldarg.1
14.    IL_0015: ldc.i4.1
15.    IL_0016: add
16.    IL_0017: ceq
17.    IL_0019: ldnull
18.    IL_001a: ldstr "Contract.Result<int>() == i + 1"
19.    IL_001f: call void System.Diagnostics.Contracts.Contract::Ensures(bool, string,
        string)
20.    IL_0024: ldloc V_0
21.    IL_0028: ret
22.} // end of method Program::Increment

```

Listing 6 Compiled method in bin\Debug\CodeContracts\assembly_name.Contracts.exe

Another question remains regarding the inner-workings of Code Contracts: what if we need to access the contracts within an assembly without forcing the user to check either static or runtime checking? The answer lies in the Contract Reference Assembly—the option to build an additional assembly especially for this purpose.

“A contract reference assembly A.Contracts.dll for assembly A contains the publicly visible interface of A along with its contracts, but no code bodies. Such contract reference assemblies are used both during rewriting to inherit contracts across assemblies, as well as during static verification to discover contracts on methods and types from other assemblies than the assembly under analysis” (15). Listing 5 shows the compiled for our example method in such an assembly.

4. Tools

a. Visual Studio Extensibility

Introduction

Visual Studio extensibility is a vast and ever-growing domain. The first APIs for extending Microsoft IDEs were released in 1995. At that time, Visual Basic 4.0 developers had the possibility of creating “Add-Ins” that could run inside the VB IDE or inside the Office Applications¹¹.

The top-level object in the Visual Studio automation object model is the DTE (Development Tools Environment¹²). Prior to Visual Studio 2005, the DTE class (and all the functionality related to it) was implemented as COM objects accessed through an OLE object library (env.olb).

Eighteen years and ten versions of Visual Studio after the initial Add-Ins, developers still use assembly-wrapped COM libraries to access the Visual Studio code automation. The first of such assemblies was Visual Studio 2005’s EnvDTE¹³. Afterwards, each new version of Visual Studio added new assemblies: EnvDTE80, EnvDTE90, EnvDTE90a and EnvDTE100. Each new library built upon the previous, adding features to the COM objects. However, no efforts have been made to upgrade the core functionality from COM to .NET libraries. Because of this, Visual Studio Extensibility is difficult and has a steep learning curve for .NET developers.

There is a plethora of options for extending Visual Studio. Developers can define macros for batch execution of commands and simple tasks; can create wizards for initializing customized project items; can develop add-ins that contain custom commands, tool windows and designers; hook up into development environment events like building and debugging to enrich user experience. Ultimately, these products can be grouped together in extensibility packages that can be placed in online repositories and upgraded through the managers in the Visual Studio Tools menu (Add-in, Library Package and Extension Manager).

In addition to the extensibility provided by the DSL Tools (described below) we are using several features like:

- parsing the active solution and retrieving built assemblies;
- using the default Visual Studio tool windows to display output information and error messages;
- inserting menu items and corresponding commands in the Tools menu and in the Solution Explorer context menu;
- identifying the selected item in the Solution Explorer and retrieving its physical file;
- creating new project items initialized with custom content and inserting them in a project folder chosen by the user;
- creating a custom tool window to interact with our designer based on Windows Forms and a custom modal window based on WPF.

¹¹ http://blogs.msdn.com/b/oscar_calvo/archive/2007/12/26/the-evil-envdte-namespace.aspx cited 18-05-2012

¹² <http://acronyms.thefreedictionary.com/DTE>

¹³ [http://msdn.microsoft.com/en-us/library/envdte\(v=vs.80\)](http://msdn.microsoft.com/en-us/library/envdte(v=vs.80))

We found a great resource for Visual Studio Extensibility in the MZ-Tools Articles Series¹⁴.

Menu items and commands

Context menu commands consist of several parts. The “PkgCmdIDList.cs” file holds the identifiers for all the commands we use to extend the Visual Studio Environment. The “SoftwareArchitectureLanguageCommandset.cs” double-derived class contains the method bodies for each command and the **GetMenuCommands()** override (to connect command events to the command identifiers). Two events are interesting for us: when the extensibility context holding the command becomes visible (**OnStatusChange**) and when the command is actually clicked on (**OnMenuChange**).

OnStatusChange serves to determine whether to enable the command.

OnMenuChange serves to carry out the action invoked by the command.

The label shown for each command is added in the “Commands.vsct” file (Visual Studio Command Table). The location of the command is also defined here: our own menu group (solution parsing command), the main context menu (“Open in Contract Editor”, “Model Visibility”, design pattern commands) and the Solution Explorer context menu (code verification command).

The Output tool window

Debug information and some of the error messages are displayed through the Visual Studio Output tool window. The information is split on various panes which are specific to processes (e.g., Build) and extensions (e.g., Code Contracts Editor Extensions).

We created an **Output** class to act as a proxy for the Output tool window. The class is a singleton and, upon initialization, it creates a new pane for the AMOC extension. Every time one of the public **Write** and **WriteLine** methods is called, the message which is passed as parameter is written to the AMOC pane.

The ErrorList tool window

Using the ErrorList tool window is a bit more complicated than using the Output one. The info, warning and error messages are all stored in a single list, each message having a unique id.

Just as for the Output tool window, we created an **ErrorList** proxy singleton class. When the singleton is initialized it instantiates an **ErrorListProvider** for the AMOC extension, and it hooks up event handlers to the solution events **BeforeClosing** and **ProjectRemoved**.

The messages are published to the ErrorList tool window through the **ErrorListProvider**. Each message is represented by an **ErrorTask** object. Errors messages can be related to various aspects of Visual Studio development. If, for example, it is a parsing error then the project, project item, line and column coordinates of the error in the code text are all saved to the **ErrorTask**.

¹⁴ http://www.mztools.com/resources_vsnet_addins.aspx

Before the **ErrorTask** is published through the provider, an event handler is hooked up to the task's **Navigate** event. When the task is double-clicked in the tool window, the **Navigate** event handler will retrieve the exact location and open it with the appropriate designer.

In our code, most of the error messages are related to the AMOC model. Although we wish to create the functionality for launching the AMOC designer when double-clicking on such an error and focusing the screen on the faulty shape, we considered this feature to have low priority and did not develop it until the moment of writing this report.

b. DSL Tools

Introduction

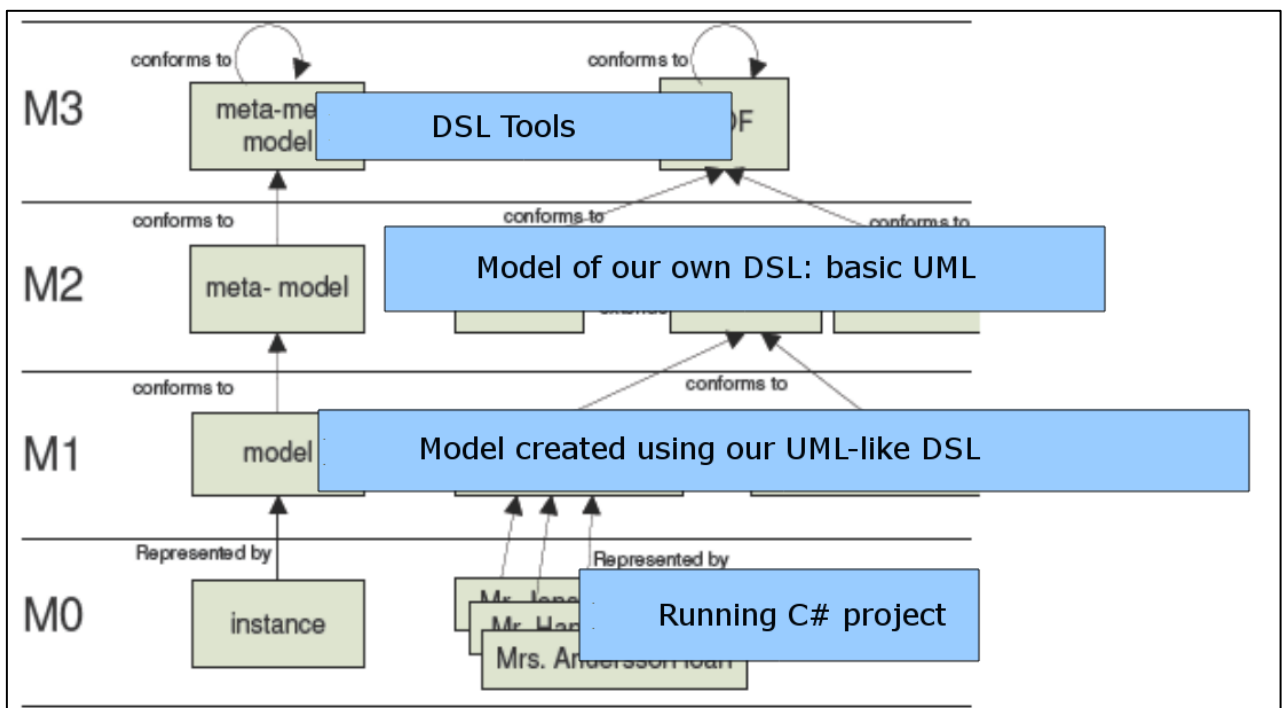


Figure 6 How AMOC is spanning across the four-layer modeling architecture by OMG. (10)

Microsoft is one of the model-driven software development tool vendors. DSL Tools is a toolkit in Visual Studio that serves the purpose of defining Domain-Specific Languages by a graphical designer and by defining code generators. The DSL Tools framework integrates with Visual Studio and makes use of Visual Studio Extensibility. Thereby it is a very important element in Microsoft's model-driven strategy. (21) DSL Tools is described in detail in the "Domain-Specific Development with Visual Studio DSL Tools" book by Steve Cook, Gareth Jones, Stuart Kent and Alan Cameron Wills. This toolkit lies on the meta-metamodel level (M3) of OMGs four-layered architecture on Figure 6. This is supported by the fact that DSL Tools are themselves generated from a DSL definition.

The next paragraph briefly describes how to start using DSL Tools. The DSL Tools book covers details about these features. (5)

Getting started with DSL Tools

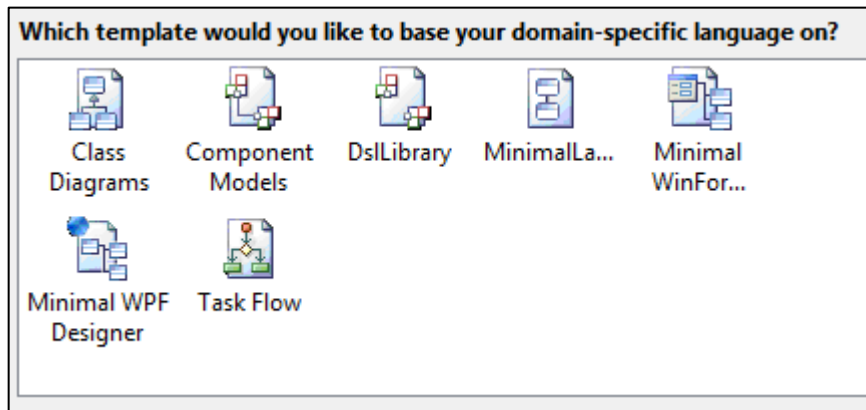


Figure 7 Titles of the different DSL Designer templates offered by the DSL Tools SDK

Visual Studio offers several templates for rendering a Domain-Specific Language Designer project (Figure 7). Based on which template is selected for the specific solution, different artifacts are generated, varying the built-in functionality. The “Minimal” templates require creating a visual diagram from scratch and so provide a lot of flexibility. An example: AMOC is based on Class Diagrams as we try to create a general conceptual model.

After choosing a template, a name for the DSL and a file extension for model diagrams, two Visual Studio projects are set up automatically: a class library containing the actual DSL definition and a package project (containing code that handles the integration with Visual Studio Extensibility and a designer template with the selected extension in the “Project Item Template” folder).

The class library contains the designer for defining the DSL. The DSL Tools SDK provides a special set of items in the Visual Studio Toolbox for adding elements in the designer and a tool window for navigating the model in a tree structure. The next paragraph briefly mentions what these tools are for.

Tools in DSL Tools

```
<?xml version="1.0" encoding="utf-8"?>
<Dsl xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion="1.0.0.0"
  <Classes>...</Classes>
  <Relationships>...</Relationships>
  <Types>...</Types>
  <Shapes>...</Shapes>
  <Connectors>...</Connectors>
  <XmlSerializationBehavior Name="SolutionArchite" Namespace="Company.Solutio">...</XmlSerializatio
  <ExplorerBehavior Name="SolutionArchitectureLanguageExplorer" />
  <ConnectionBuilders>...</ConnectionBuilders>
  <Diagram Id="5458aa4c-b317-4" Description="Description for" Name="SolutionArchite" DisplayName="N
  <Designer CopyPasteGeneration="CopyPasteOnly" FileExtension="sa" EditorGuid="fbd53bb1-5344-4">...
  <Explorer ExplorerGuid="5cbae8c3-e17f-4" Title="SolutionArchite">...</Explorer>
</Dsl>
```

Figure 8 Editing “DslDefinition.dsl” as an XML file

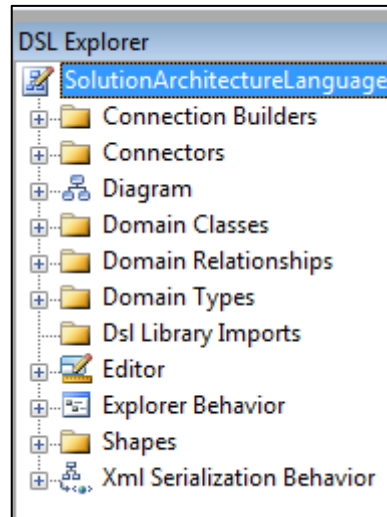


Figure 9 Structure of DslDefinition.dsl in the DSL Explorer toolwindow

The most important file in a DSL Tools solution is “DslDefinition.dsl”. This XML file can be edited in three different ways: through an XML or text editor, through the DSL Designer or through the DSL Explorer (from the DSL Tools SDK). The most expressive way is through the XML editor, but it also gives way to inconsistencies (Figure 8). However, using the Designer and the DSL Explorer (Figure 9) gives a better visual overview and implies that the restrictions of the language are automatically imposed.

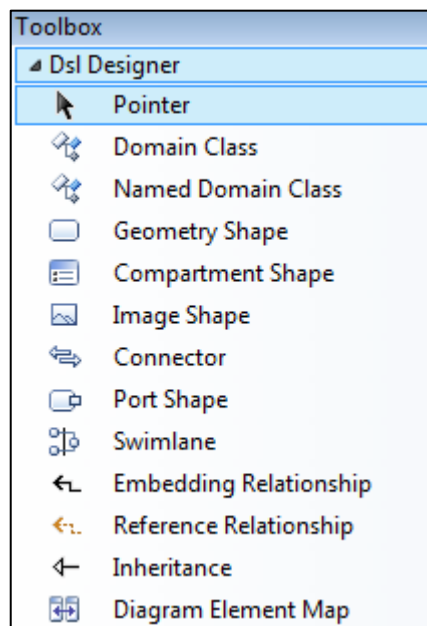


Figure 10 Domain elements, graphical elements and their mapping in the DSL Tools toolbox of Visual Studio 2010

The Visual Studio Toolbox provides items for both “Classes and Relationships” and “Diagram Elements” when we work on a DSL using the DSL Designer (Figure 10). The following represent tools for specifying problem domain elements: Domain Class, Named Domain Class, Embedding

Relationship, Reference Relationship and Inheritance. The rest of the tools in the Toolbox are presentation elements. In order to make use of the graphical presentation elements in the model, a mapping has to be set up using the Diagram Element Map tool. Such a mapping is shown as a thin connector line between a graphical element and the problem domain element.

The *Diagram* element is the basic placeholder or parent of all shapes and connectors. There is only one single instance of a *Diagram* in a running designer, thus it is not present among the Toolbox items.

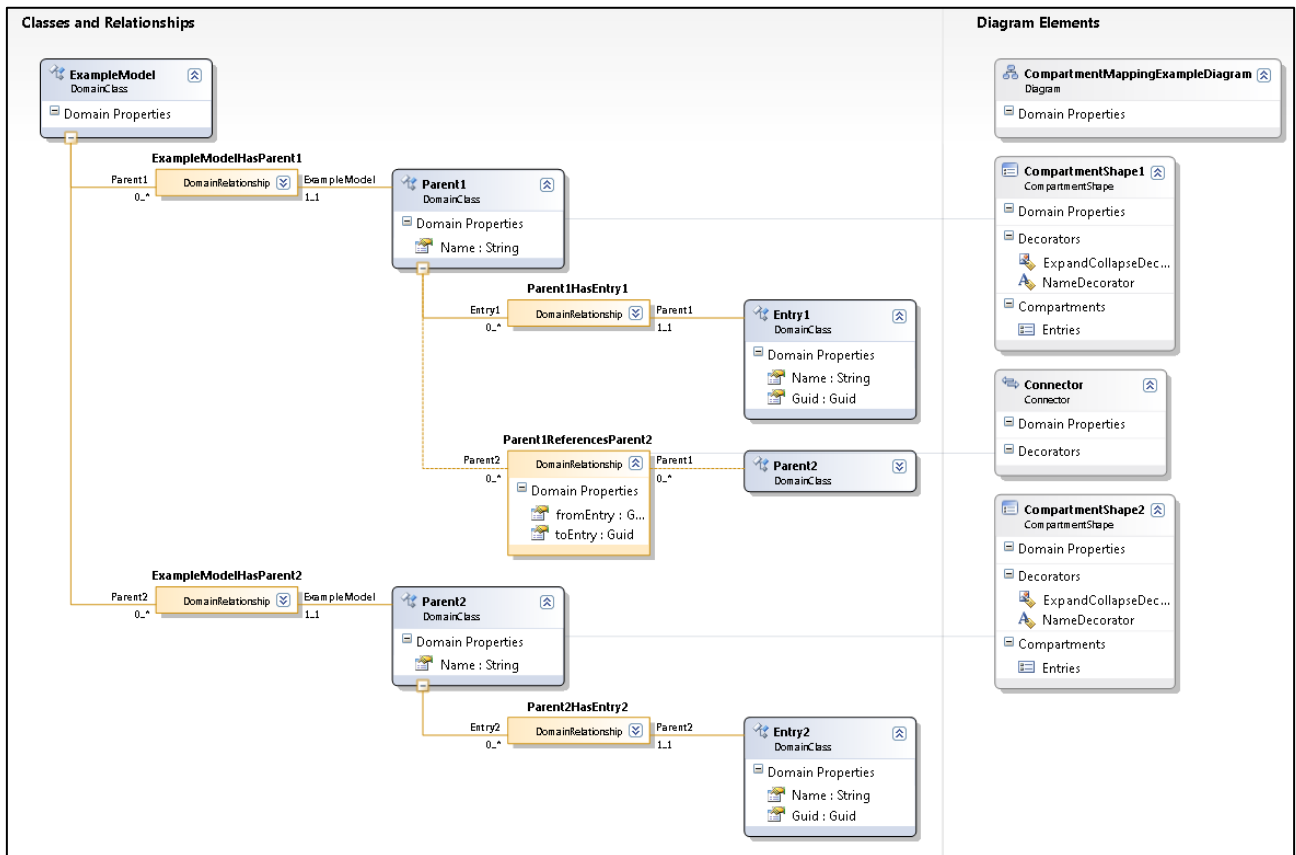


Figure 11 Example of a DSL definition in DslDesigner

The DSL Explorer tool window is only open if the DSL Designer is open too. It shows a tree view of the XML with edit functionality through a context menu, and provides access to all the XML tags present in “DslDefinition.dsl”.

DSL Designer is a visual designer that has two columns (Figure 11), one for “Classes and Relationships” and one for “Diagram Elements”. This is an important separation of context. Developers can create various graphical representations of a model without changing a bit in the underlying problem domain model.

Everything present in the “Classes and Relationships” column is a mapping to the problem domain of the desired DSL in question. There are domain classes and domain relationships that are supposed to specify the problem domain. The collection of these “defines just the concepts dealt with by the DSL” (5).

The “Diagram Elements” column contains the presentation model of the DSL (the graphical definition). Basically, we talk about boxes and lines. There are three basic kinds of shapes in DSL Tools: *GeometryShape*, *CompartmentShape* and *Connector*. Shapes are connected through connector *LinkShapes* and/or contain embedded graphical model elements to visualize the domain relationships and classes.

Transforming templates

Tool	Graphical element class from SDK
Geometry Shape	NodeShape
Compartment Shape	NodeShape
Image Shape	NodeShape
Port Shape	NodeShape
Connector	LinkShape
Domain Class	ModelElement
Named Domain Class	ModelElement
Embedding Relationship	ElementLink
Reference Relationship	ElementLink - LinkShape
Inheritance	ElementLink - LinkShape

Table 1 Results of transforming presentation elements by the “Transform All Templates” command

One step that cannot be left out when creating a DSL extension is to transform the DSL definition to platform specific code that is compiled into an assembly for the resulting Visual Studio Extension. The DSL SDK installs by default to the Visual Studio installation folder and it contains C# text templates with “.tt” extension. The XML in DslDesigner is transformed to generated C# code using these text templates. This transformation is carried out by the command “Transform All Templates” (Table 1). As a general rule, the generated code should not be modified directly but through one of the 3 ways described above for editing “DslDefinition.dsl”.

As we use the generated code to achieve our goals from section 1.d, it is required to know about the object hierarchy of DSL Tools. In the DSL Tools SDK, the graphical elements and the domain classes both are subclassing **Microsoft.VisualStudio.Modeling.ModelElement**. As stated in the publication “The Object Constraint Language. Getting your models ready for MDA” by Jos Warmer and Anneke Kleppe: “Everything in a model is a **ModelElement**” (7). This principle holds when creating a DSL for Visual Studio just as much as for UML. Interestingly, the “**ModelElement**” property on *NodeShapes* references the concrete domain object (an instance of a defined DomainClass).

As one of our goals is to give users the ability to model their existing code, we use the Common Compiler Infrastructure (described below) to parse assemblies.

c. CCI - Common Compiler Infrastructure

The oldest reference to a common compiler infrastructure was made in 2003 when Barend H. Venter’s “Multi-language compilation” patent was filed (22). However, it was only in March 2009 that the two CCI projects (“Metadata Components” and “Code and AST Components”) became

available on CodePlex¹⁵. This bit of historical information is relevant in connection to the fact that we did not find any published research papers that mention CCI in any form. Hence, all following information comes from online API documentation and personal experience.

Possible usage scenarios include:

- “writing a custom static analyzer operating on assembly metadata or IL;
- rewriting assembly metadata or IL;
- generating IL and metadata;
- using CCI as a managed replacement for the IMetadata interfaces.”¹⁶

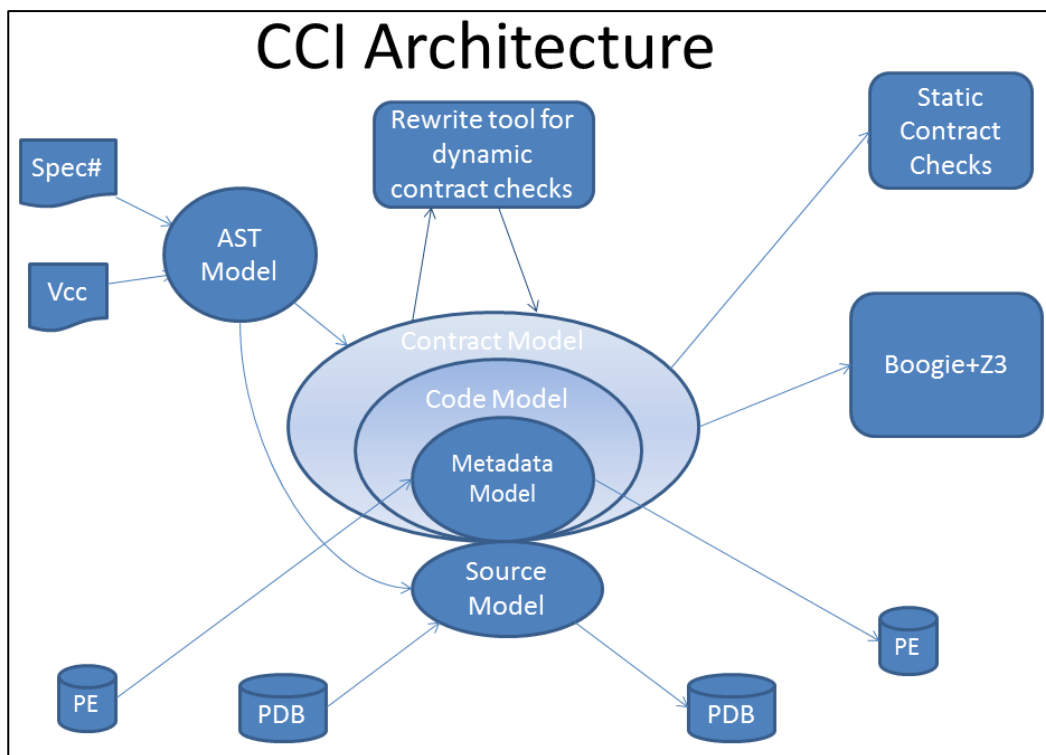


Figure 12: CCI Architecture¹⁷

CCI's purpose is to provide an integrated set of components (Figure 12) for compiling high-level languages for the Common Language Runtime (CLR) and perform post compilation operations on the resulting ASTs. The main components are: Metadata API, Code Model API, and AST API.

The CCI Metadata API provides functionality for reading, analyzing and writing .NET assemblies, modules and debugging (PDB) files. It is similar to .NET **System.Reflection** and **System.Reflection.Emit** APIs, but contains additional functionality and has better performance.

¹⁵ <http://ccimetadata.codeplex.com>, <http://cciast.codeplex.com>

¹⁶ <https://cciast.svn.codeplex.com/svn/Documentation/QuickIntroductionToCCI.docx>

¹⁷ <https://ccimetadata.svn.codeplex.com/svn/Documentation/CCI2%20Architecture.pptx>

The Code Model API is built on top of the Metadata API. This API provides functionality for working with the program architecture as the architecture is built in the assembly. The entire model tree of the application can be traversed and analyzed by it.

The AST API builds upon previous APIs to add method bodies to the Code Model tree. As the AST structure maps to CIL instructions, CCI is not very useful when dealing with C# or other .NET languages (and it is not its purpose to be). A C# source emitter exists in the CodePlex test samples, but CCI does not include a high-level language parser. We used CCI for parsing assemblies, extracting contracts and building the model.

d. Project Roslyn

Project Roslyn is similar to CCI in many respects. It also attempts to provide compiler, AST and program analysis functionality to developers. Similarly to CCI, Roslyn is a young research project with great focus on the development and release of the product. However it is not well covered by published scientific papers. Except for the Roslyn project overview (23), we mainly rely on online walkthroughs and personal experience to describe it.

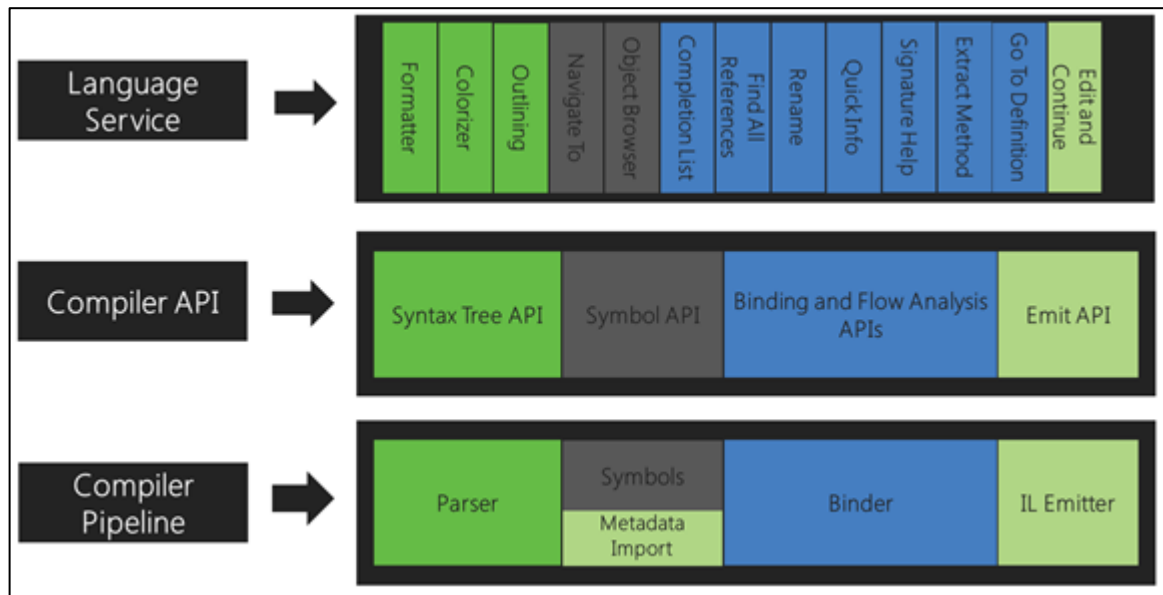


Figure 13 Roslyn architecture (23)

Roslyn's goal is to rewrite the C# and VB.NET compilers in the C# and VB.NET languages respectively. It also aims to expose the compiler functionality to developers through an API. The compiler API follows the compilation process pipeline step by step: syntax tree representation, symbols and semantics, binding referenced assemblies and emitting IL. Additionally, it provides a set of language services that facilitate usage of the compiler API. Finally, by installing Roslyn, one gets access to Visual Studio extensibility templates for developing code editor issue fixes, code completion providers etc. We used Roslyn to analyze and modify contracts in the model (section 5).

5. AMOC features

a. Visualization of code class architectures in VS2010

i. Introduction

There is a clear need for visualizing software architectures. When thinking of software architectures as networks, algorithms from the igraph software package described in (24) provide full-blown circle, sphere and various other layouts, with the promise of efficiency even in case of million shapes and connectors, thus providing scalable and easy to understand graphs.

Very interesting ideas emerge from 3D visualizations of software architectures, too. Loe Feijs and Roel de Jong (authors of the publication “3D Visualization of Software Architectures”) describe LEGO-shaped architecture visualization (25). EvoSpaces from “EvoSpaces: 3D Visualization of Software Architecture” by Sazzadul Alam and Philippe Dugerdil represents software architectures and code metrics as a city map (26). Similar ideas are described in the Foreword of the DSL Tools book called “Domain-Specific Development with Visual Studio DSL Tools” by Steve Cook, Gareth Jones, Stuart Kent and Alan Cameron Wills. (5).

ii. Class diagram in Visual Studio DSL Tools

Our work is focusing on the problem domain of a class diagram that supports Design By Contract through the use of Code Contracts. A class diagram is a static structure of a model with classes and types, internal structures and their relationships (8). Class diagrams are logical views of the software architecture. (27)

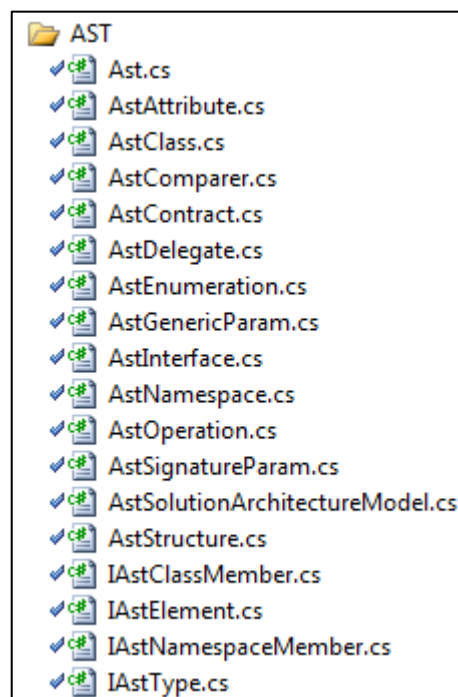


Figure 14 Parts of the AST, with helper interfaces and classes included

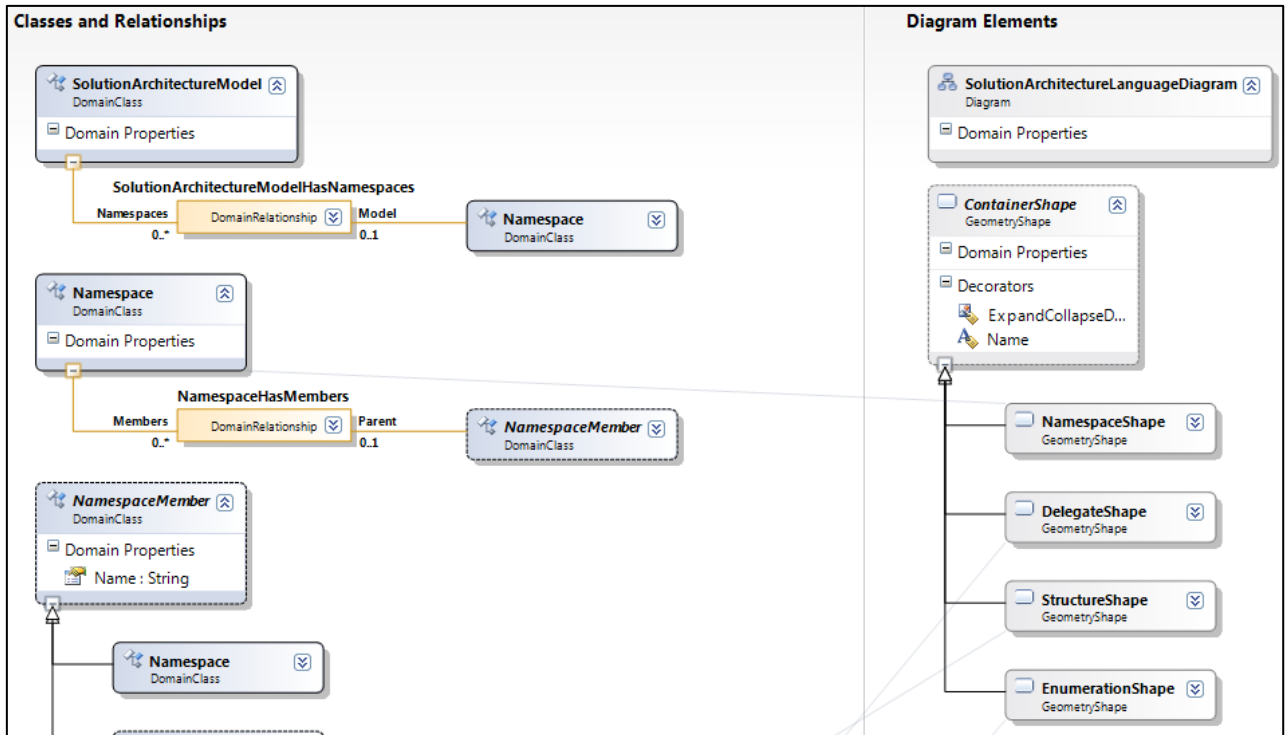


Figure 15 First quarter of AMOC's DSL definition

Our class diagram consists of ModelElements. All these elements are being mapped to a DomainClass in the DSL Designer (Figure 15 - Column for "Classes and Relationships"). An "Ast" prefix is added to the names of DomainClasses when we parse existing code for building a model as described in section 5.h (Figure 14). The topmost element of the AMOC model is called "SolutionArchitectureModel". It should have another name as to show the relation to the conceptual model our problem domain, namely a mixture of basic UML and C#. The following model element names originate from UML:

Attribute - "An attribute defines values that can be attached to instances of the class or interface."¹⁸ *Attributes* are the UML alternative of C# Fields¹⁹. Our model visualizes the Name and Type of an *Attribute*.

Operation - "An operation is a method or function that can be performed by an instance of a class or interface". *Operation* is the UML alternative of C# Method²⁰. Our model visualizes the following from *Operations*: Contracts, Name, Generic Parameters, Return Type, and Signature (by Signature Parameters). Class constructors and Properties are modeled in our solution as *Operations*.

Operation Contract - In UML, an Operation Contract is a condition that specifies the system state before and after the operation executes²¹.

¹⁸ <http://msdn.microsoft.com/en-us/library/dd323861.aspx>

¹⁹ <http://msdn.microsoft.com/en-us/library/ms173118>

²⁰ [http://msdn.microsoft.com/en-us/library/ms173114\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/ms173114(v=vs.100).aspx)

²¹ <http://msdn.microsoft.com/en-us/library/dd323859.aspx>

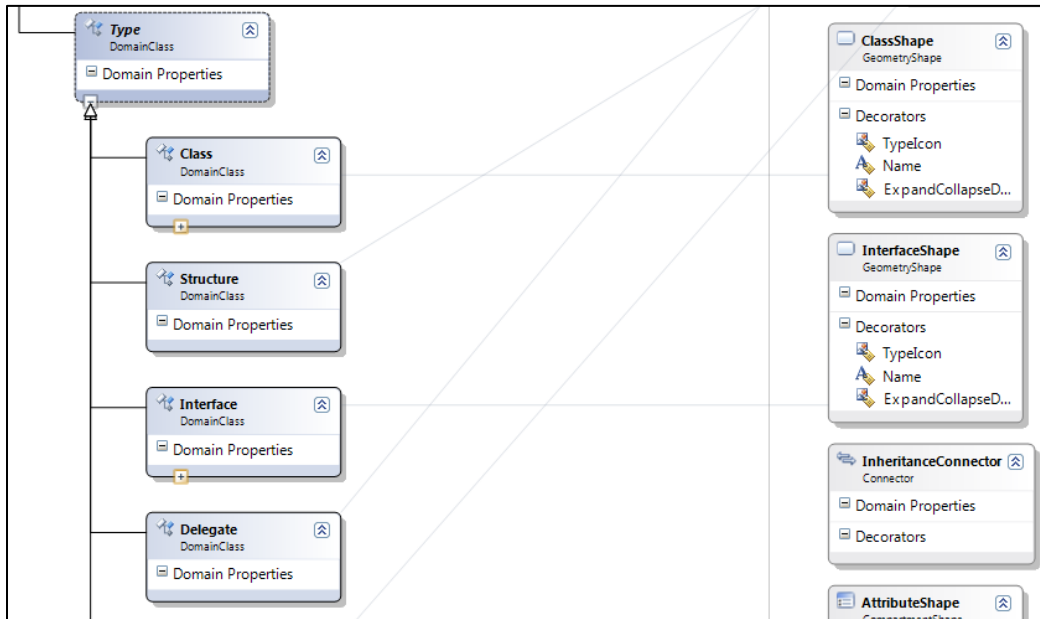


Figure 16 Types in the AMOC definition

The name for the rest of the AMOC model elements originates from C#: *Class*, *Interface*, *Enumeration*, *Delegate*, *Struct*, *Namespace*. The top model element, “SolutionArchitectureModel” serves as the parent of every Namespace. This embedding relationship is named “SolutionArchitectureModelHasNamespaces”. *Namespaces* have the feature of embedding other *Namespaces* or *Types*. *Types* are the following DomainClasses: *Class*, *Structure*, *Interface*, *Delegate*, and *Enumeration* (Figure 16)

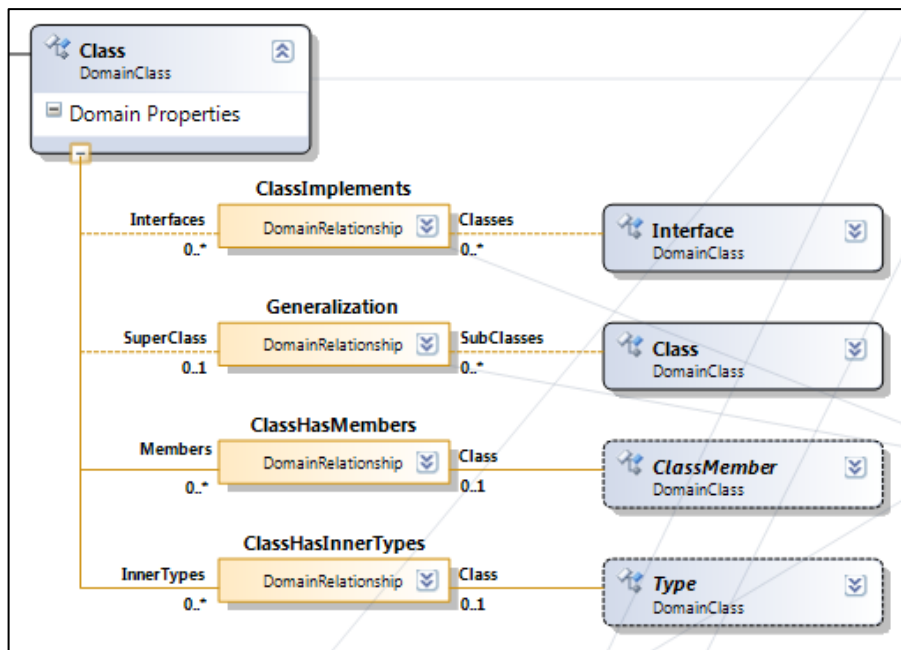


Figure 17 Relationships of Classes

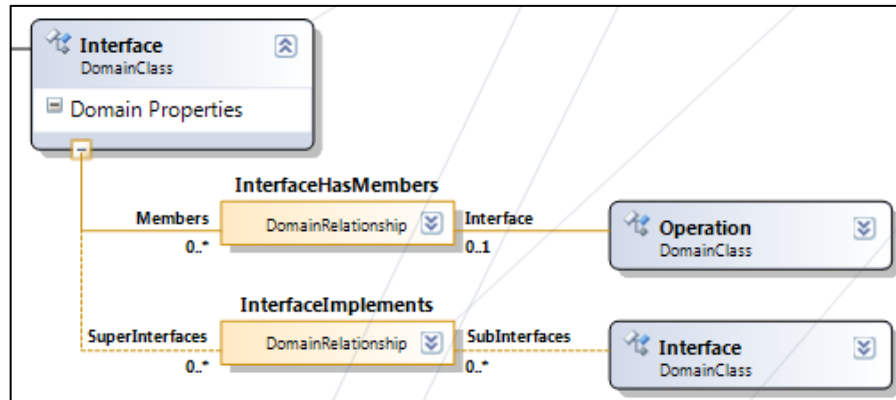


Figure 18 Relationships of Interfaces

Figure 17 and Figure 18 show the relationships related to *Class* and *Interface* domain classes expanded. An *Interface* has two relationships: embedding *Operations* and inheriting from other *Interfaces*. A *Class* has the ability to:

- implement *Interfaces*;
- embed *ClassMembers* (Operations and Attributes);
- inherit from a single *SuperClass*;
- reference inner *Types*.

The next paragraph explains which presentation elements AMOC uses in connection with which logical or domain element.

iii. Graphical elements in AMOC

As the column for “Diagram elements” in Figure 15 shows, there are a number of graphical tools associated with our designer to represent the mapped DomainClasses: a diagram, GeometryShapes, Connectors and CompartmentShapes.

All GeometryShapes represent a ModelElement with an area; therefore they generate NodeShapes when the DSL Tools “Transform All Templates” command is run.

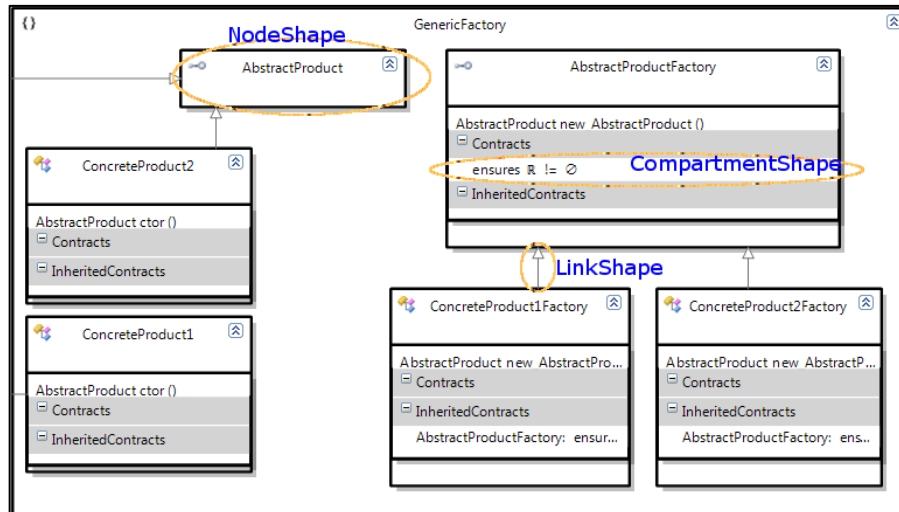


Figure 19 Example for NodeShape, CompartmentShape, LinkShape

NodeShape is a shape with an area in the diagram. It is used to represent the following domain classes from our code architecture: *Namespace*, *Class*, *Interface*, *Structure*, *Enumeration*, *Delegate*, *Attribute* and *Operation*.

A *CompartmentShape* is also a rectangular shape. It serves as means to visualize embedding relationships - used for *Contracts*.

A *LinkShape* is an arrow, vector or line. It is used to represent relationships such as inheritance between superclass and subclass or an *Interface* and the class implementing this *Interface*.

Figure 19 shows a working example for these three basic shapes in the context of an abstract factory. The highlighted *NodeShape* is an *Interface* called “AbstractProduct”, the *CompartmentShape* is the “**Contract.Ensure(Contract.Result())!=null;**” *Contract* in its short-handed form (described in section 5.d), while the *LinkShape* is an *InheritanceConnector* (“ConcreteProduct1Factory” is an implementation of the “AbstractProductFactory” *Interface*). The Abstract Factory design pattern is described in section 5.g.

Bounds of a *NodeShape* are the shapes relative X, Y coordinates (relative meaning counting from the top and left side of a parent shape), width and height. Bounds are in effect a *RectangleD* structure that stores a set of a 4 doubles - location and size of the rectangle. This structure is a basic GDI+ shape. The .NET Framework uses the GDI+ library for rendering graphics, an example being the Netron Graph Library Architecture described by Francois M. Vanderseypen. (28)

iv. Layout possibilities

DSL diagrams support nesting child shapes. (5) This nesting has been used originally to show all the embedding relationships in the AMOC model (*Operations* in a *Class* or *Interface*; *Types* in *Namespace*s). Our extension of the basic *NodeShape* is the *ContainerShape*. A *ContainerShape* ensures that nested child shapes don't position outside the bounds of their parent by resizing the parent. We have discovered the following options for positioning nested child shapes in a *ContainerShape*:

- letting the shapes render on top of each other: this way the latest rendered *NodeShape* has the possibility of completely hiding the underlying shapes;
- using the built-in function **NodeShape.AutoLayoutShapes** (referred to as **AutoLayout** further);
- calculating bounds of shapes ourselves.

	Autolayout	Custom grid layout
4 shapes	1s	1s
30 shapes	1s	1s
140 shapes - SolutionArchitectureLanguage	5s	1s

Table 2 Performance for expanding a single NamespaceShape with different number of child shapes in the designer. Test configuration: Visual Studio 2010 Experimental Instance running as Debug mode for the SolutionArchitectureLanguage solution. Time measured between invoking the expansion with a double click on the collapsed NamespaceShape and seeing the fully expanded NamespaceShape with all its directly nested child shapes.

AutoLayout re-generates the layout of shapes inside a container. The shapes have a completely new position upon the execution of **NodeShape.AutoLayoutShapes**. Our experiences also show that the scalability of this built-in algorithm is questionable, as positioning the shapes this way can render the UI thread of the AMOC designer busy for several seconds (Table 2). Its only benefit is that the nested child shapes are laid out without position calculation.

We have customized the layout of *NodeShapes* in the following scenarios:

- opening an AMOC diagram;
- double clicking a *NodeShape*, namely: *NamespaceShape*, *ClassShape*, *InterfaceShape*;
- expanding and collapsing a *NodeShape*;
- rendering *OperationShapes* in *ClassShape* and *InterfaceShape*.

v. Customizing the layout of nested child shapes

All *NodeShapes* are collapsed by default in AMOC. These shapes have a predefined size. **AutoLayout** lays out the *NodeShapes* in a grid with offset (Appendix 1). We have achieved a similar layout using our grid layout algorithm. A grid is a trivial chessboard-like layout described for example in the EvoSpaces publication. (26)

Our steps in laying out *NodeShapes* in a grid are the following:

1. we find out the measures of the grid;

2. if the grid should have a square-like ratio for side height and width (as the **AutoLayout** function lays it out), that means we are interested in the square root of the *NodeShape* count;
3. the resulting grid will have equal sized rows of and a number of equal sized columns—this presents a certain overhead in the area covered by the *NamespaceShape* when the square root of the shape count is not an integer.

Example: If there are 46 shapes to fit in a grid, the square root is 6.7823. The sides are found by the following logic: starting at 6, increase the sides by one till the product of the smaller side and the longer side does not exceed 46.

* $6 \times 6 = 36 < 46$

* $6 \times 7 = 42 < 46$

* $7 \times 7 = 49 > 46$

```

1. var columns = Convert.ToInt32(Math.Floor(Math.Sqrt(childNodeShapes.Count)));
2. var rows = columns;
3. var availablePositionCount = rows * columns;
4. var rowIncreased = false;
5. while (availablePositionCount < childNodeShapes.Count)
6. {
7.     if (!rowIncreased)
8.     {
9.         rows++;
10.        rowIncreased = true;
11.    }
12.    else
13.    {
14.        columns++;
15.        rowIncreased = false;
16.    }
17.    availablePositionCount = rows * columns;
18.}

```

Listing 7 Initial algorithm for finding out the row and column count

The cost of this algorithm is the loop on Listing 7. An alternative to avoid the cost of using a **while()** loop (line 5-18) is to rounding up the square root to the closest integer. However, one should be aware of this possible overhead: The likelihood for more wasted spaces increases by the number of shapes.

The square root of a count is a .

The integer achieved by rounding down ad . Rounding a up gives au .

The closer is a to ad than au , the bigger the wasted space will be.

Example: There are 150 shapes needing a layout in a *NamespaceShape*. a will be 12,25. ad will be 12. Following the algorithm on Listing 6, we get an end measure of $12 \times 13 = 156$. 156 already is enough poles in the grid (6 wasted places). Rounding up in this case gives 13 rows and columns.

$13 \times 13 = 169$, meaning 19 wasted places when we need to layout only 150. 19 is more than a whole row (or column) of wasted positions, in theory.

```

1. foreach (var childNodeShape in childNodeShapes)
2. {
3.     if (currentColumn >= columns)
4.     {
5.         //new row needed.
6.         currentColumn = 0;
7.         currentRow++;
8.     }
9.     //calculate the XY position of the nodeshape based on current row and column.
10.    var X = currentColumn * columnwidth + offsetX;
11.    var Y = currentRow * rowheight + offsetY;
12.    childNodeShape.Bounds = new Rectangled(X, Y, shapewidth, shapeheight);
13.    currentColumn++;
14.}
15. var calcWidth = offsetX + columns * columnwidth;
16. var calcHeight = offsetY + currentRow * rowheight;

```

Listing 8 Positioning algorithm. *calcHeight* is the final height of the *NamespaceShape*

On the other hand, the algorithm positioning each *NodeShape* on Listing 8 only increases the height of the *NamespaceShape* based on the number of actually used rows, so in the end only 12 rows will be used from 13, resulting in the more acceptable overhead of 6 spaces. This positioning algorithm leaves the empty space visible at the bottom corner of the last row in **Error! Reference source not found..** In some cases however, the overhead could be a row containing a single element while the other rows each contain 10 shapes: The space for a new row of nested child shapes becomes reserved in the *NamespaceShape* already when the first shape is positioned to take the starting position in the row.

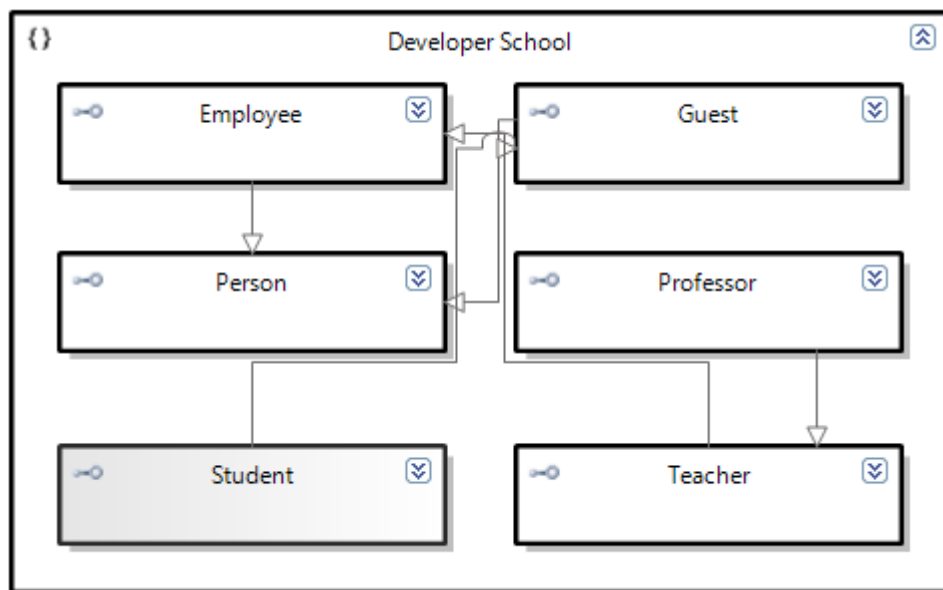


Figure 20 Positioning model elements only by their Name

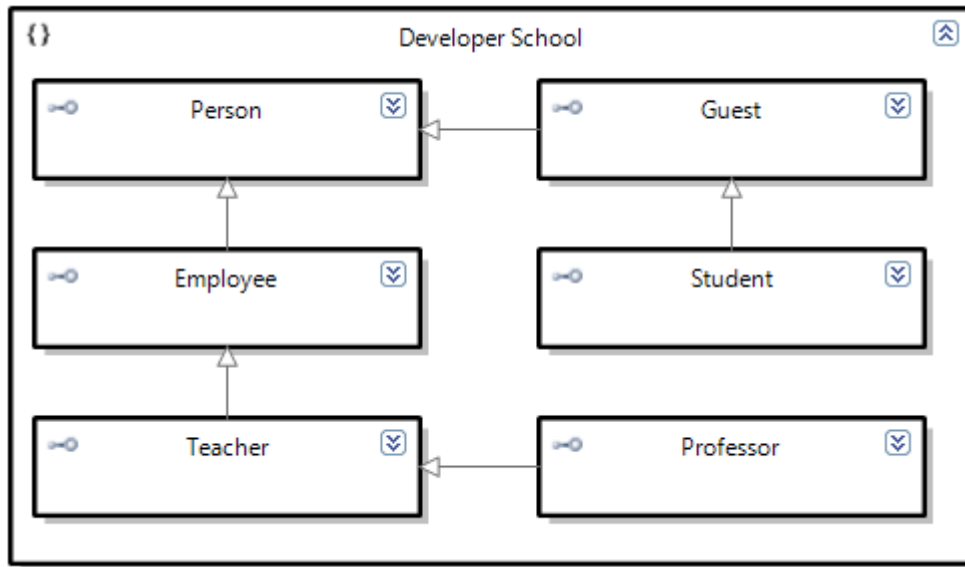


Figure 21 Positioning model elements by how closely they are related

As *LinkShapes* follow the *NodeShapes* in the diagram automatically, it is also useful to group related shapes close to each other in the grid. For now we order the shapes by their names before positioning them to rows, as similar names can indicate relationships in the model elements. It would be however possible to sort these elements based on how close the relationships really are to the first element of model element group: directly connected (single or no link between two *ModelElements*), 1 step (related through one *ModelElement*), 2 steps ... n steps. An example is shown on Figure 20 for the current positioning of *ModelElements*, and the same *ModelElements* positioned by the number of connections to “Person” on Figure 21. On this trivial example the Person is the most abstract Interface and has only two subinterfaces connected. However, if an interface has one or more extra connectors pointing to it, it already makes sense to position it in the center of the graph. Different layout possibilities (chessboard, concentric circles) are mentioned in the publication about *EvoSpaces* (26). Further research in graph layouts would be necessary in order to implement efficient and neat class diagrams for AMOC.

vi. Arranging shapes in the same *NamespaceShape*

As using our own way to position nested child shapes is possible on *NodeShape* Expand or Collapse, so it becomes straightforward to push down shapes which are present in the same parent shape and would be partially or completely covered when resizing a *NodeShape*.

There are three parts of this functionality: the action for pulling up shapes, the action for pushing shapes down and the action to find all the relevant shapes positioned below.

As we know the shapes are positioned in a grid, it is not enough to find below shapes based on their **Bounds.Y** property (the relative position from top inside a parent). They should be in the same column too. **Error! Reference source not found.** shows the **FindBelowShapes** method.

b. Model Visibility

Modeling software systems means that the designer can easily become cluttered with elements. Also, there are situations when only a snapshot of the model is required to focus on a specific issue.

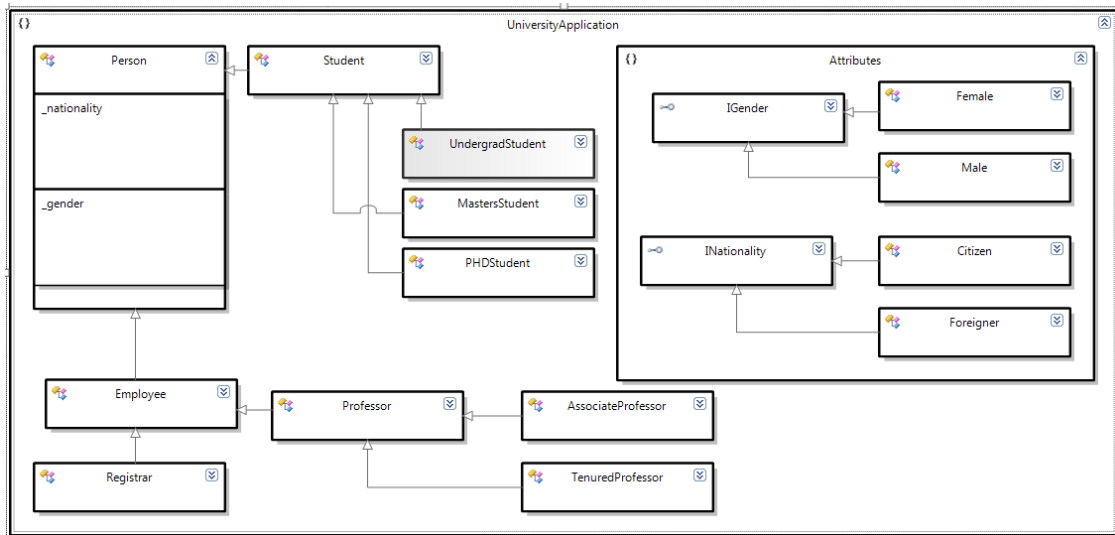


Figure 22 Model Visibility example: University Application

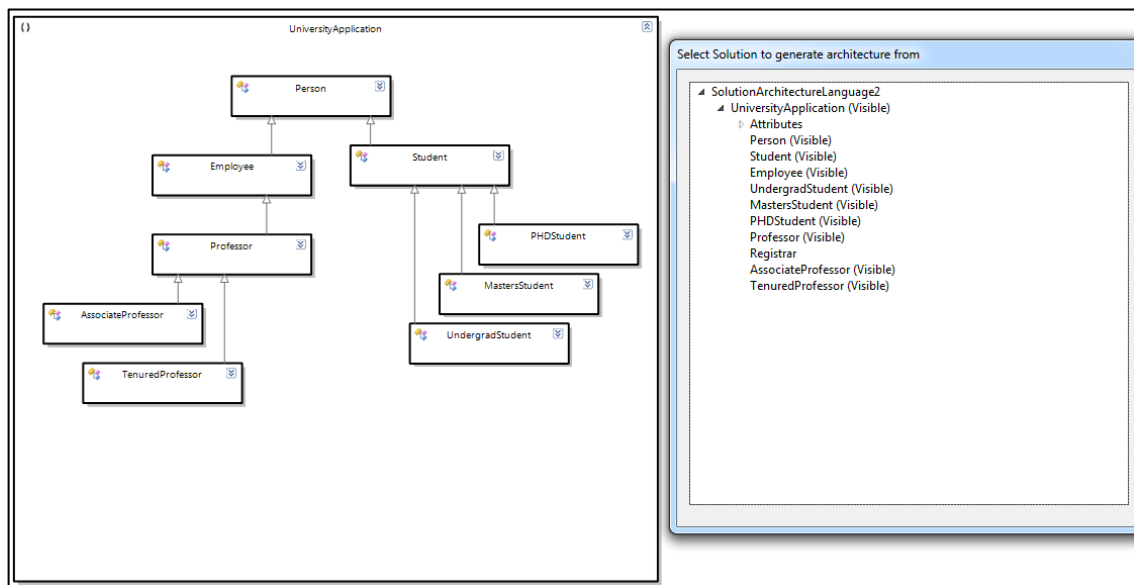


Figure 23 Model Visibility dialog window

Such an example is the University Application in Figure 22. While it is a very small model, it is big enough to fill the entire designer window. Let us assume that we want to emphasize the Student and the Professor classes and their immediate related elements.

When selecting “Model Visibility” from the designer’s context menu, we are presented with a dialog window where we can set the visibility of each model element (Figure 23). Visibility is set by using each shape’s Hide and Show methods (predefined by DSL tools).

c. Contract editor

Various methods for contract-editing

This chapter briefly goes through all the options of editing a Code Contract in our modeling tool. A direct, not inherited Code Contract of an *Operation* can be edited by:

- in the *CompartmentShape* of an *OperationShape* in the designer diagram, showing the contract expression;
- in the Properties window of Visual Studio;
- using the text input fields of an Operation tab in the Contract Editor window.

Editing the contract directly in the designer diagram and in the Properties window is completely managed by DSL Tools. In these two cases there is no IntelliSense support or any other kind of suggestion. IntelliSense is a productivity feature of Visual Studio providing syntax completion of logically related code elements from a dropdown menu when writing code, e.g., in the Text Editor of Visual Studio²².

The text input fields in a Contract Editor Tab support showing a suggested list of related code elements. The next chapter is focusing on the features of our Contract Editor tool.

Introduction

This chapter is about how we display and manage contracts in our own Visual Studio tool window. Contract Editor has the following general features:

- Opening a ModelElement by showing information about it in a tab. ModelElements in this case are instances of a subset of the DomainClasses listed in section 5.b, namely instances of Classes, Interfaces, Operations. This means there are 3 different tab layouts available: Class, Interface, Operation.
- Editing ModelElement information with the help of text suggestions for the text input fields on a tab. Suggestions come from the ModelElements that share the Namespace with the selected ModelElement and are displayed as a list when typing.
- Saving the updated ModelElement information back to the model by a “Save” button. This action has to be executed within a modeling transaction.
- Grouping the opened ModelElements by their relationships in colored tabs. Each time a user opens a ModelElement, the ModelElement’s tab header will be colored. If there are any ModelElements already open in the Contract Editor that the newly added ModelElement is a direct child or parent of, the new tab is added on the right side of the tab showing the related ModelElement
- Multiple ModelElements can be passed to the Contract Editor simultaneously. The designer diagram allows selecting multiple ModelElements when holding the Ctrl key on the keyboard.
- Keeping track of ModelElements opened from multiple designer diagrams of the same Visual Studio instance. Each time a user opens a ModelElement from a designer that has not yet been used in the Contract Editor, it will be open on the first and only tab in

²² <http://msdn.microsoft.com/en-us/library/43f44291.aspx> cited on 23-05-2012

the Contract Editor. The `ModelElement` is also added to the temporary collection related to the open designer. When the user returns to a previously opened designer and opens a `ModelElement` from there, this temporary collection of `ModelElements` is restored to the Contract Editor tabs.

- “Close” button to remove the tab from Contract Editor and to remove its `ModelElement` from the temporary collection related to the open designer.
- Validating a selected `ModelElement` when clicking the “Validate” button on the tab having focus (Not implemented)
- Editing the Name of the `ModelElement` (Not implemented)

Features by `DomainClass`

The following describes what features are available in a Contract Editor Tab depending on which `DomainClass` does the opened `ModelElement` belong to. Only the features not mentioned in the general feature list above are discussed here.

Interface tabs offer:

- Listing the *Operations* of the *Interface*. Clicking an *Operation* opens it in a new Operation tab of Contract Editor
- Editing Interface Attributes

Class tabs offer:

- Listing the *Operations* of the *Class*. Clicking an *Operation* opens it in a new Operation tab of Contract Editor
- Editing Class Attributes

Operation tabs offer:

- Editing *Signature* of the *Operation*
- Editing *Operation Contracts*. The input text field for Contract expressions adds the following code elements to the text suggestions: “**ensures**, **requires**, **Contract**, **Contract.Result()**”

Implementation details

Our Contract Editor tool is in fact a Windows Forms User Control. Its code files “`UserControl1.cs`” and “`CustomToolWindow.cs`” are located in the `DslPackage` project. The class `CustomToolWindow` is a subclass of the `Microsoft.VisualStudio.Modeling.Shell.ToolWindow` base class. `CustomToolWindow` uses `UserControl1` as layout. The Contract Editor’s main part is a `TabControl` that displays a tab for each `ModelElement` that is passed to it by the “Open in Contract Editor” designer context menu command.

Grouping ModelElements

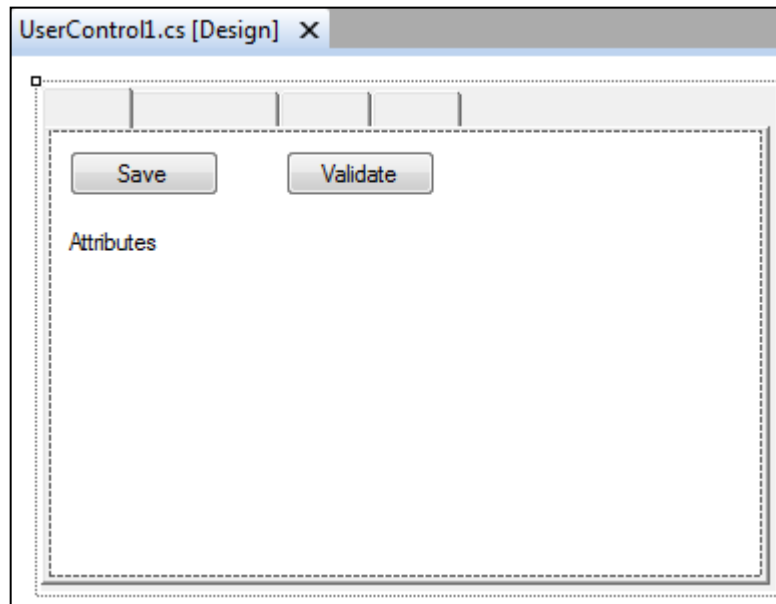


Figure 24 The UserControl in Design view, showing predefined tabs

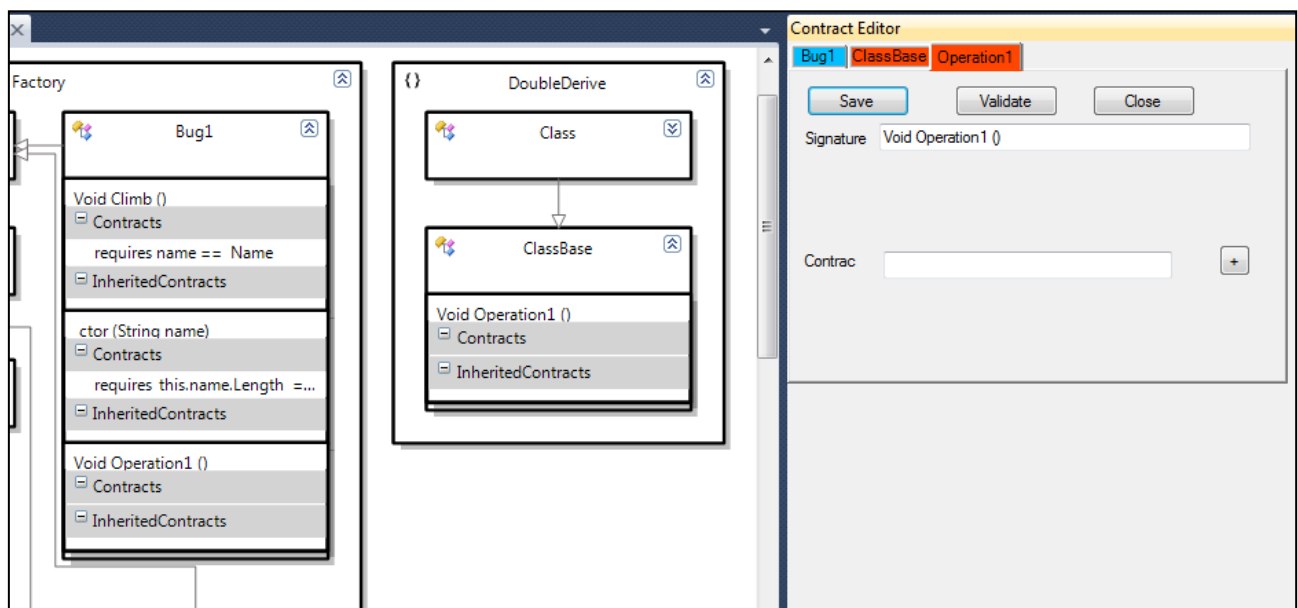


Figure 25 Grouped Class “ClassBase” and its Operation “Operation1” in the Contract Editor

By default, the tab headers do not provide a property for changing the background color. To change the background color of tab headers, the `Windows.Forms.TabControl`'s `DrawMode` was set to `OwnerDrawFixed` (from `Normal`) (Figure 24). Therefore the tab headers are not visible in the design view of UserControl.

Groups of ModelElements must have distinct colors. At the time of this writing, the colors red, yellow and blue are used. Parent and child relationships are the model's *InterfaceHasMembers*

and *ClassHasMembers* relationships (described in section 5.a). The example *Class* “ClassBase” and its *Operation* “Operation1” shown in Figure 25 are connected by a *ClassHasMembers* relationship.

Dropdown code element suggestions

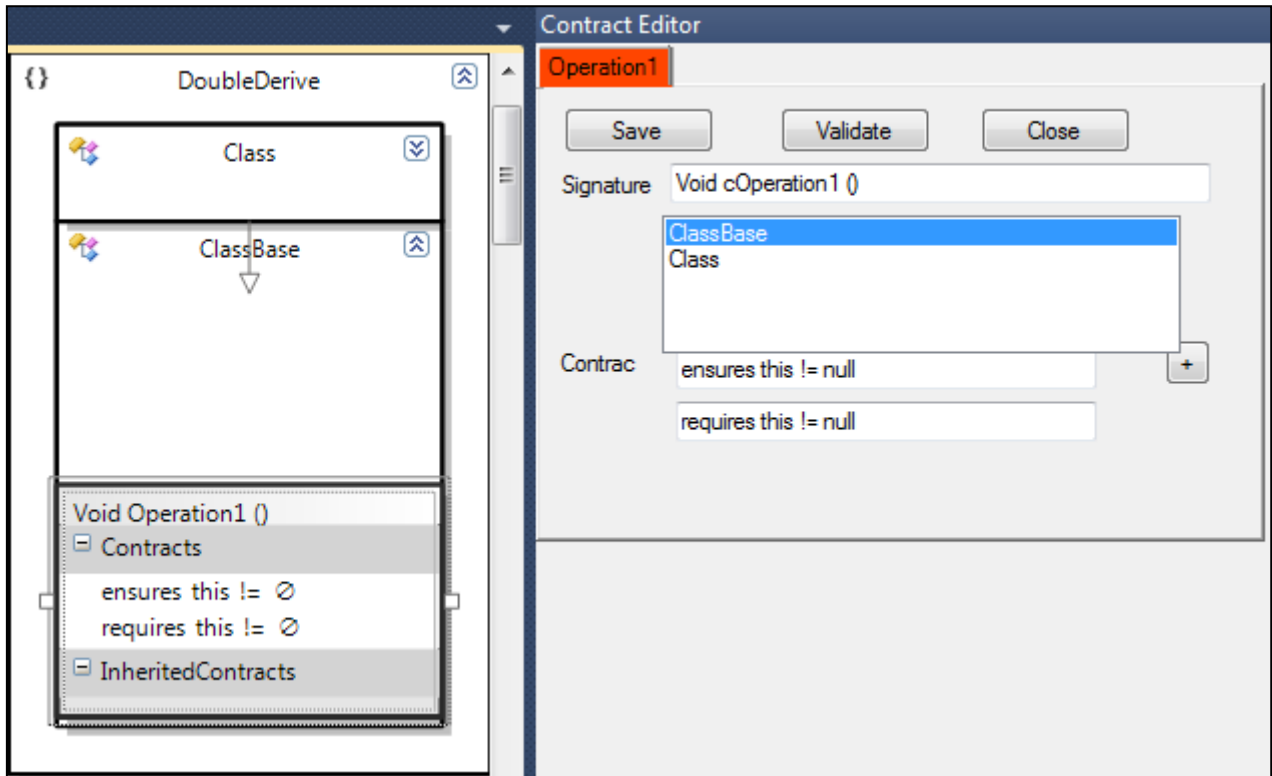


Figure 26 “ClassBase” and “Class” suggested when typing a “c” in the signature of the Operation “Operation1”

The basis of this feature is a recursive function that populates a list for contract suggestions and a list for editing *ModelElement* Names and *Operation* signatures.

Code elements for the list come only from *ModelElements* sharing the *Namespace* with the selected *ModelElement* displayed on the open Contract Editor tab. These code elements are limited to the following strings (as the solution is only a proof-of-concept):

- Name of *Class*, *Operation*, *Attribute* *ModelElement*
- “this”, “base”

The code elements included in the contract suggestions list are all the code elements from the standard list and the following strings: “ensures ”, “requires ”, “Contract”, “Contract.Result()”.

On each **KeyUp** event (`Windows.Forms.Control1.KeyUp`) inside an input text field (`Windows.Forms.TextBox`) we look for a non-empty string appearing before the current cursor position and the last space character. This string will be the lookup string.

The dropdown is shown when we find one or more strings in the suggestion list that begins with the lookup string (Figure 26). The left side of Figure 26 suggests that “ClassBase” and “Class” code elements came from the shared *Namespace* “DoubleDerive”. The lookup string gets replaced by a suggestion if the user clicks on a suggestion or presses Enter. The Up and Down keys navigate between the suggested code elements in the list. The suggestion list disappears if there are no suggestions, the user clicks outside the TabPage or when the lookup string is replaced.

d. Shorthand Contracts

Motivation

The inspiration for creating shorthand contracts came from the BON method (29). BON uses two forms of specification – graphical and textual – which implies two ways of writing contracts. While the textual specification uses a set of characters that is easily handled by most text editors, the graphical specification employs mathematical symbols. This means that one can easily write textual BON from a keyboard without spending time in search for special characters, while the contracts take less space on screen.

1. Result -> (single and other.single and other not member_of children and other not member_of parents and sex /= other.sex)
2. Result -> (single and other.single and other \notin children and other \notin parents and sex \neq other.sex)
- 3.
4. for_all c member_of children it_holds (exists p member_of c.parents it_holds p = Current)
5. $\forall c \in \text{children} \bullet (\exists p \in c.\text{parents} \bullet p = @)$

Listing 9 Comparison of BON assertions

In Listing 9 we show two examples of BON assertions (a post-condition and an invariant). Lines 1 and 4 show the assertions in the longer, textual notation. It is obvious that the graphical notation on lines 2 and 5 is more suitable for a visual designer.

Approach

C# / Code contracts	Shorthand	Graphical	Textual BON	Explanation
		Δ name	delta name	Attribute changed
Contract.OldValue(p)	p'	old expr	old expr	Old return value
Contract.Result	\mathbb{R}	Result	Result	Current query result
this	@	@	Current	Current object
null	\emptyset	\emptyset	Void	Void reference
+ - * /	+ - * /	+ - * /	+ - * /	Basic numeric operators
		\wedge	\wedge	Power operator
/	/	//	//	Integer division
%	%	\\	\\	Modulo
==	==	=	=	Equal
!=	!=	\neq	/=	Not equal
<	<	<	<	Less than
<=	<=	\leq	<=	Less than or equal
>	>	>	>	Greater than
>=	>=	\geq	>=	Greater than or equal
		\rightarrow	->	Implies (semi-strict)
		\leftrightarrow	<->	Equivalent to
!	!	\neg	not	Not
&&	&&	and	and	And (semi-strict)
		or	or	Or (semi-strict)
^	^	xor	xor	Exclusive or
Contract.Exists	\exists	\exists	exists	There exists
Contract.ForAll	\forall	\forall	for_all	For all
			such_that	Such that
	•	•	it_holds	It holds
	\in	\in	member_of	Is in set
		\notin	not	Is not in set
		: type	: type	Is of type
		{ }	{ }	Enumerated set
		Closed range

Table 3 Assertion elements in BON with C# / Code contracts equivalents

Looking at the list of assertion elements in BON (Table 3) we picked a few examples to demonstrate the feature.

We started by replacing the current object and the void reference with @ and \emptyset , respectively.

For the current query result we opted for the symbol \mathbb{R} to avoid confusion with the identifier *Result* that might be used for variables, properties etc.

For the old return value of an expression we suffixed the expression with an apostrophe. The reason is that the BON notation would create confusion when inserted in a C# expression (e.g. `"Count == old Count + 1"`).

The assertion elements "there exists", "for all", "it holds" and "is in set" do not have 1-to-1 equivalents in code contracts but they are used with quantifier methods as follows:

```
1. bool ForAll<T>(IEnumerable<T> collection, Predicate<T> predicate)
2. bool Exists<T>(IEnumerable<T> collection, Predicate<T> predicate)
3.
4. Contract.ForAll(strlist, s => string.IsNullOrEmpty(s))
5. Contract.Exists(strlist, s => string.IsNullOrEmpty(s))
6.
7.  $\forall s \in \text{strlist} \bullet \text{string.IsNullOrEmpty}(s)$ 
8.  $\exists s \in \text{strlist} \bullet \text{string.IsNullOrEmpty}(s)$ 
```

Listing 10 Shorthand quantifiers; case 1

Case 1: Given the quantifier methods on lines 1 and 2 (Listing 10), each takes as parameters a collection and a predicate (parameter and statement block). We can consider that the statement block must hold *for all / at least one* parameter that is a member of the collection. Therefore lines 4 and 5 are rewritten as lines 7 and 8.

```
1. bool ForAll<T>(int fromInclusive, int toExclusive, Predicate<int> predicate)
2. bool Exists<T>(int fromInclusive, int toExclusive, Predicate<int> predicate)
3.
4. Contract.ForAll(0, strlist.Count, i => string.IsNullOrEmpty(strlist[i]))
5. Contract.Exists(0, strlist.Count, i => string.IsNullOrEmpty(strlist[i]))
6.
7.  $\forall i \mid 0 \leq i < \text{strlist.Count} \bullet \text{string.IsNullOrEmpty}(\text{strlist}[i])$ 
8.  $\exists i \mid 0 \leq i < \text{strlist.Count} \bullet \text{string.IsNullOrEmpty}(\text{strlist}[i])$ 
```

Listing 11 Shorthand quantifiers; case 2

Case 2: Given the quantifier methods on lines 1 and 2 (Listing 11), each takes as parameters two integers and a predicate (parameter and statement block). We can consider that the statement block must hold *for all / at least one* parameter that is a member of the set of integers bound by the two integer parameters. Therefore lines 4 and 5 are re written as lines 7 and 8.

We did not work on shorthanding other elements for various reasons: "attribute changed", "implication" and "equivalence" do not have Code Contracts counterparts; changing most operators would only confuse developers.

Implementation

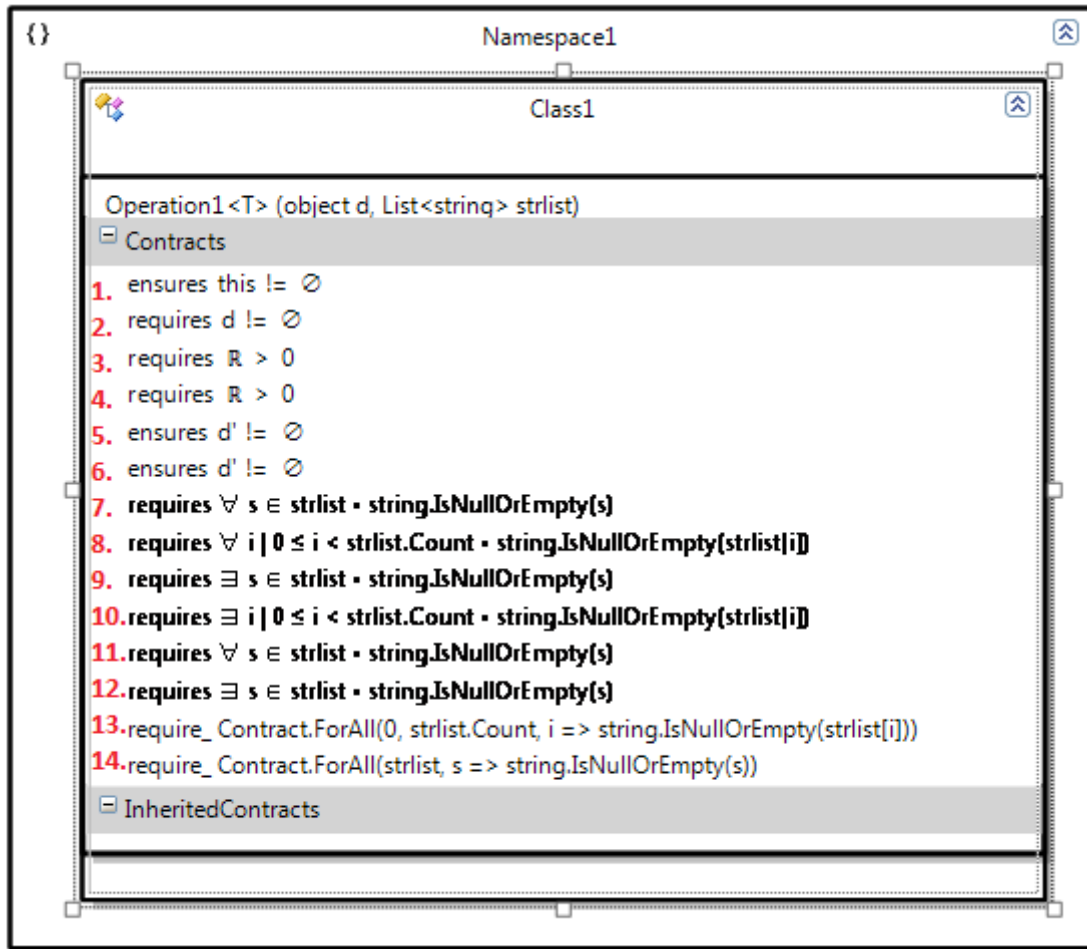


Figure 27 Shorthand contracts

As each contract assertion can be parsed as a Roslyn **ExpressionSyntax**, we process it with a recursive method that traverses the expression tree and checks for the possibility of using a shorthand contract.

Some of the shorthanded elements can be identified as simple expressions: the current object is a **ThisExpressionSyntax** object; the void reference is a **LiteralExpressionSyntax** object with the **ValueText** property set to **null**.

However, the Code Contracts methods are **InvocationExpressionSyntax** objects. The type of contract is retrieved from the object's **Expression** property. The test we make is to see if the expression string starts with one of the contract method identifiers: **Contract.ForAll**, **Contract.Exists<T>**, **Contract.Result<T>**. This way we also handle the case of generic overloads. Since the shorthand contract does not display the generic parameter we can use the same approach that we use for non-generic overloads: line 3 in Figure 27 is the shorthand of “requires **Contract.Result()** > 0” while line 4 is the shorthand of “requires **Contract.Result<int>()** > 0”.

If the contract method is **Contract.Result** we simply return the shorthand contract symbol \mathbb{R} (lines 3 and 4).

If the contract method is **Contract.OldValue** we know there is only one argument so we return it followed by an apostrophe (lines 5 and 6).

If the contract method is **Contract.ForAll** or **Contract.Exists** there are two similar cases for each.

Case 1: If the invocation object has two arguments then we know they are a collection and a predicate (lines 7, 9, 11, and 12).

Case 2: If there are three arguments then we know they are the lower and upper bounds of the integer interval for which the predicate (third argument) must hold. From this point there is only a matter of associating elements with shorthand symbols and rearranging them to obtain the shorthand contract (lines 8, 10).

Unexpected behavior

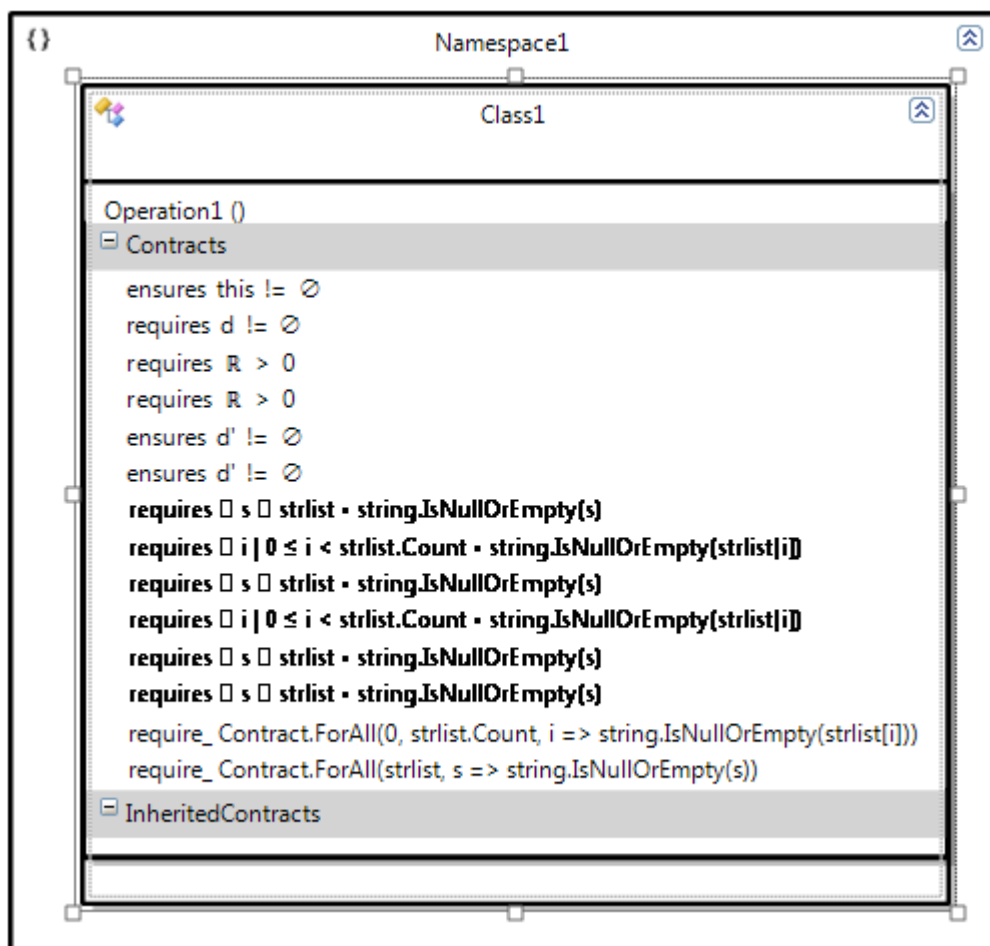


Figure 28 Shorthand contracts unexpected behavior

At one point during the development process, the DSL designer stopped showing the Unicode characters “∀”, “∃” and “∈” (Figure 28). Also, the font weight of the lines containing these characters has always been bold. Up to the moment of writing this report, we were not able to determine the cause of this behavior.

e. Contract inheritance

Contract inheritance is one of the most complex aspects of Code Contracts. When writing contracts for classes inheriting multiple interfaces, it is difficult to anticipate the exact behavior of the runtime checker. To get some information on where are inherited contracts coming from, the developers can use the Visual Studio Editor Extensions which display the contracts as code adornments.

Our extension attempts to take this one step further by displaying all the elements involving one set of contracts at once. For demonstration purposes, we have implemented this feature for Class elements. Figure 29 shows a contract inheritance example (the colored markers are not part of the designer).

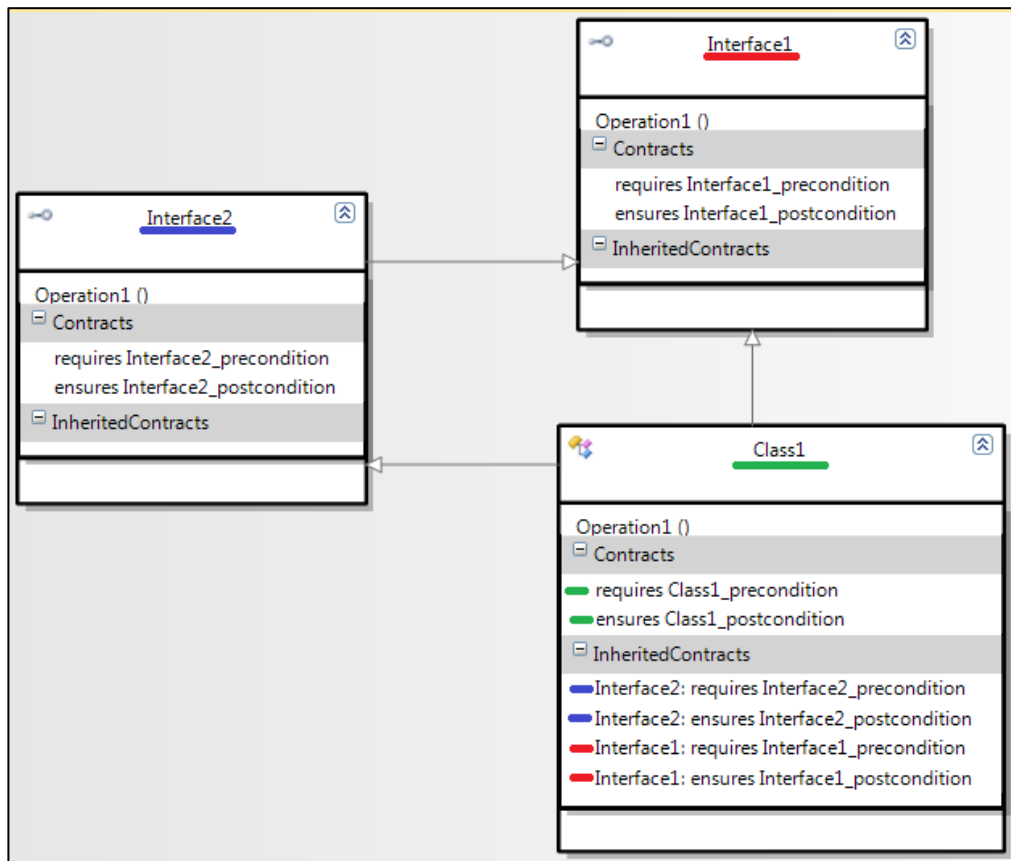


Figure 29 Contract inheritance examples

The *OperationShape* is inheriting from the *CompartmentShape*. We have defined a compartment for the element’s own contracts and one compartment for inherited contracts. When the *InheritedContracts* compartment is displayed for an *OperationShape*, we get the reference for the *Operation*’s parent *Class*, loop through all the *Interfaces* that it implements, for

each *Interface* we look for an *Operation* with the same signature and append its contracts to our compartment.

This approach allows users to see where each contract is defined. Another benefit is that the contracts in the *Interface* elements and the ones in the *InheritedContracts* compartment reference the same elements so changing one automatically updates the other.

The downside is that the actual list is only computed when the *ClassShape* loads, which means that adding a contract to an *Interface Operation* does not update the *InheritedContract* compartments of that *Operation's* implementations. This can be done manually by iterating through all the *Classes* that implement that specific *Interface*.

An extra feature of the designer is that when it is saved, each *Operation* will check whether it inherits contracts from an *Interface Operation*. Even if the *Interface Operation* has no preconditions it still counts as the most general precondition possible: the **true** value. In this case the *Operation* will iterate through its own contracts and warn the user of the existence of local preconditions that break the Liskov substitution principle: “*Namespace1\Class1\Operation1\ requires Class1_precondition: Local precondition is not allowed because it is strengthening inherited preconditions.*”.

Other inheritance issues

Further research of Code Contracts inheritance revealed that the behavior of the static and runtime checkers is inconsistent. While the runtime checker verifies all inherited contracts, the static one is throwing warnings in several cases.

```

[ContractClass(typeof(ContractsForIOBJECTWithWheels))]
public interface IOBJECTWithWheels
{
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    void InstallWheels();
}

[ContractClassFor(typeof(IOBJECTWithWheels))]
public abstract class ContractsForIOBJECTWithWheels : IOBJECTWithWheels
{
    public void InstallWheels()
    {
        Contract.Requires("IOBJECTWithWheels precondition" == string.Empty);
        Contract.Ensures("IOBJECTWithWheels postcondition" != string.Empty);
    }
}

[ContractClass(typeof(ContractsForICar))]
public interface ICar : IOBJECTWithWheels
{
    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    new void InstallWheels(); // new does not make a difference
}

[ContractClassFor(typeof(ICar))]
public abstract class ContractsForICar : ICar
{
    public void InstallWheels()
    {
        Contract.Requires("ICar precondition" != string.Empty);
        Contract.Ensures("ICar postcondition" != string.Empty);
    }
}

abstract class Limousine : ICar
{
    //public virtual void InstallWheels() ...

    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    public abstract void InstallWheels();
}

class HummerLimousine : Limousine
{
    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    public override void InstallWheels()
    {
        Contract.Requires("HummerLimousine precondition" != string.Empty);
        Contract.Ensures("HummerLimousine precondition" != string.Empty);
    }
}

```

Listing 12 Inheritance example (abstract)

Let's consider the example in Listing 12. We want to decide which the inherited contracts for the **HummerLimousine** class are. **HummerLimousine** is a subclass of the abstract class **Limousine** which implements the interface **ICar** which, in turn, implements **IOBJECTWithWheels**.

In each of the classes and interfaces we specify contracts for the **InstallWheels** method. We also use the Visual Studio Editor Extensions to display the inherited contracts.

ContractsForIObjectWithWheels.InstallWheels defines two contracts for **IObjectWithWheels.InstallWheels**. The contracts are visible in the adornment.

ContractsForICar.InstallWheels defines two contracts for **ICar.InstallWheels**. The two contracts and the ones inherited from **IObjectWithWheels.InstallWheels** are visible in the adornment. The runtime checker will test all four conditions. However, the static checker throws a warning: *“Contract class InheritanceTest.ContractsForICar cannot define contract for method InheritanceTest.IObjectWithWheels.InstallWheels as its original definition is not in type InheritanceTest.ICar. Define the contract on type InheritanceTest.IObjectWithWheels instead.”*

In this case neither pre- nor postconditions should be defined in the **ICar** interface, which is strange because it should be allowed to add a postcondition that would strengthen the postcondition of the superinterface. Even if the warning is displayed, all the contracts are checked at runtime and this inconsistency adds to the confusion.

The **Limousine** class implements the **InstallWheels** method as abstract. We get the following warning: *“Method 'InheritanceTest.Limousine.InstallWheels' cannot implement/override two methods 'InheritanceTest.ICar.InstallWheels' and 'InheritanceTest.IObjectWithWheels.InstallWheels', where one has Requires.”* The warning comes from Code Contracts, not the compiler and, even though it says “cannot implement”, the code compiles fine. It is impossible to prevent the method to implement the interface because it is legal in C#. It would make sense to display the message as an error instead. The question is which is the correct behavior? Are the contracts inherited or not?

```

[ContractClass(typeof(ContractsForIOBJECTWithWheels))]
public interface IOBJECTWithWheels
{
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    void InstallWheels();
}

[ContractClassFor(typeof(IOBJECTWithWheels))]
public abstract class ContractsForIOBJECTWithWheels : IOBJECTWithWheels
{
    public void InstallWheels()
    {
        Contract.Requires("IOBJECTWithWheels precondition" == string.Empty);
        Contract.Ensures("IOBJECTWithWheels postcondition" != string.Empty);
    }
}

[ContractClass(typeof(ContractsForICar))]
public interface ICar : IOBJECTWithWheels
{
    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    new void InstallWheels(); // new does not make a difference
}

[ContractClassFor(typeof(ICar))]
public abstract class ContractsForICar : ICar
{
    public void InstallWheels()
    {
        Contract.Requires("ICar precondition" != string.Empty);
        Contract.Ensures("ICar postcondition" != string.Empty);
    }
}

abstract class Limousine : ICar
{
    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    public virtual void InstallWheels()
    {
        Contract.Requires("Limousine precondition" != string.Empty);
        Contract.Ensures("Limousine precondition" != string.Empty);
    }
    //public abstract void InstallWheels();
}

class HummerLimousine : Limousine
{
    |requires "Limousine precondition" != string.Empty
    |requires "ICar precondition" != string.Empty
    |requires "IOBJECTWithWheels precondition" == string.Empty
    |ensures "Limousine precondition" != string.Empty
    |ensures "ICar postcondition" != string.Empty
    |ensures "IOBJECTWithWheels postcondition" != string.Empty
    public override void InstallWheels()
    {
        Contract.Requires("HummerLimousine precondition" != string.Empty);
        Contract.Ensures("HummerLimousine precondition" != string.Empty);
    }
}

```

Listing 13 Inheritance example (virtual)

The answer comes from the **HummerLimousine** where contracts from both interfaces are present. Once more, all the contracts are checked at runtime.

Listing 13 shows exactly the same interfaces as Listing 12. The difference is that the **Limousine** class implements the **InstallWheels** method as virtual. In this case we get a different warning: “Method 'InheritanceTest.Limousine.InstallWheels' implements interface method 'InheritanceTest.ICar.InstallWheels', thus cannot add Requires.” This time it is allowed to implement the interface method and to inherit the contracts. However, it is not allowed to add preconditions which is in accordance to the Liskov substitution principle (a new preconditions would strengthen the inherited contract; this is not allowed). Still, this is only a warning; the runtime checker tests all three preconditions (inherited and local).

What is even more confusing is that the **HummerLimousine** method inherits all contracts, but Code Contracts does not throw a warning due to the fact that it defines its own precondition. In this case it is still a breach of the Liskov principle, but because the class inherits from a superclass and not an interface the breach is ignored.

f. Contract optimization

Contracts optimization is a generic term for our idea of statically analyzing the contracts in the model and output errors, warnings and suggestions for improving the contract quality. The aspects that we tried to improve are:

- expression style: suggest ways of rewriting the expression in a more useful form;
- undefinedness: pointing out expressions that may evaluate to null resulting in a null reference exception instead of a more meaningful contract exception.

To perform this kind of analysis we required access to the contract’s abstract syntax tree and, if possible, semantics.

One of our initial ideas was to model the expression tree of each contract, but we soon realized that we would not be able to do this in the time allocated for this project. Also, expanding the graphical designer to handle full expression trees would have made it cumbersome and counter-productive as class diagram. We decided to represent contract expressions as plain text strings and perform optimizations and other transformations through pattern matching.

When we later revisited the idea of optimizing contracts, we had already come across Roslyn. With the Roslyn API we were able to easily parse the contract expressions and obtain an abstract syntax tree.

Roslyn can also resolve symbols from the AST, but to do this it requires parsing the equivalent of a full program and the referenced assemblies. If we were to access the semantics of the contracts in the context of the model, we would have to generate C# code for the entire model and process it with Roslyn. However, we did not have time to develop the model-to-code transition and so we were only able to perform our analysis on the abstract syntax.

Validating the expression style

Validating the expression style was our first experiment with Roslyn. While there are plenty of guidelines for the C# code style, they focus on aspects like source file organization, indentation, comments, statement, naming rules etc. However, there are no guidelines of writing style for expressions.

In our search for an example (to form the base of a proof-of-concept), we turned to compiler optimizations.

Such an example is constant folding. Given the expression `cardsCount <= (2 + 3)`, the addition is an unnecessary run-time operation that can be easily avoided. Additionally, 2 and 3 are “magic numbers”. If the expression is part of a Poker game system it makes sense that the number of cards in a hand is at most 5, which is mathematically equivalent to `2 + 3`, but semantically the numbers 2 and 3 have no real meaning. This is a situation that can be detected by analyzing the AST. When encountering a `BinaryExpressionSyntax`, we can test whether the operator is `“+”` or `“-”` and the two operands have the `LiteralExpressionSyntax` type. If the two literals can be parsed as integers we make a suggestion to the user to replace the subexpression with their sum or difference.

Another example is removing Boolean literals. Given the expression `isRound == true & isGreen == false` we can easily remove the literals and still keep the meaning, but in a more compact form: `isRound & !isGreen`. Again, it is a matter of evaluating binary expression operands and testing whether they are Boolean literal expressions.

The possibility of analyzing and improving the contract expression style exists and has a great potential. But we have to keep in mind the target of these optimizations. Contracts should be as meaningful for the developer as they are for the compiler. Let us consider the example of the common subexpression elimination: given the expression `“(a + b) + (a + b) * 20 / 100”`. This expression can be optimized by computing a temporary variable first (`aplusb = a + b`) and using it in the expression. A static checker might even be smart enough to suggest the extraction of the common factor: `aplusb * (1 + 20 / 100)` resulting in fewer operations for the compiler.

But if the formula in the expression has a meaning in itself, reducing it for the sake of the compiler might confuse the developer. Consider the surface area of a cylinder:

- `Surface_Area = Areas_of_top_and_bottom + Area_of_the_side;`
- `Surface_Area = 2 * (Area_of_top) + (Perimeter_of_top) * height;`
- `Surface_Area = 2 * (Pi * r * r) + (2 * Pi * r) * h.`

If a static checker suggests evaluating `“2 * Pi * r”` in advance and extracting the common factor then the result remains the same but the formula is losing meaning:

- `_2PIr * (r + h).`

Checking for undefinedness

The question that inspired us to check for undefined expressions is: if Code Contracts are verifying code execution then who is verifying Code Contracts execution? Let's consider the case in Listing 14 **Error! Reference source not found.**

```
1. string GetLine(List<string> lines, int i)
2. {
3.     Contract.Requires(lines != null);
4.     Contract.Requires(0 <= i & i < lines.Count);
5.
6.     return lines[i];
7. }
```

Listing 14

The two preconditions eliminate the possibility of getting null reference or index out of bounds exceptions. Both preconditions are checked before the actual method body is executed and they work as a whole. We can consider that all preconditions of a method are bound by conditional AND operators: "**lines != null && 0 <= i & i < lines.Count**". When one of the conditions evaluates to **false**, the rest are ignored, and a meaningful exception is thrown. What if the preconditions are written like in Listing 15 or 16?

```
1. Contract.Requires(0 <= i & i < lines.Count);
2. Contract.Requires(lines != null);
```

Listing 15

```
1. Contract.Requires(lines != null & i < lines.Count);
2. Contract.Requires(0 <= i);
```

Listing 16

In both cases **lines.Count** is evaluated even if **lines** is **null**.

If we consider each condition to be a mathematical function, then its codomain is the set of truth values (true and false) and its domain is the Cartesian product of the sets containing possible values for all variables in the condition. The problem occurs when those variables can be null meaning that they are not defined.

One of the theoretical approaches to solving this problem is to add a third constant to the truth values for the undefined (\perp) (30). However, this breaks the associativity of the equivalence relation (\equiv) between truth values.

Since every truth value is equivalent to itself it holds that $(\perp \equiv \perp) \equiv \text{true}$. And since the equivalence relation is associative and symmetric it also holds that $(\perp \equiv \text{true}) \equiv \perp$ and $(\text{true} \equiv \perp) \equiv \perp$. The question is what should $(\perp \equiv \text{false})$ evaluate to? All three options lead to breaking the associativity of the equivalence relation.

- if $(\perp \equiv \text{false}) \equiv \text{false}$ then $((\perp \equiv \text{false}) \equiv \text{false}) \equiv \text{true} \neq \text{false} \equiv (\perp \equiv (\text{false} \equiv \text{false}))$
- if $(\perp \equiv \text{false}) \equiv \text{true}$ then $((\perp \equiv \text{false}) \equiv \text{false}) \equiv \text{false} \neq \text{true} \equiv (\perp \equiv (\text{false} \equiv \text{false}))$
- if $(\perp \equiv \text{false}) \equiv \perp$ then $((\perp \equiv \perp) \equiv \text{false}) \equiv \text{false} \neq \text{true} \equiv (\perp \equiv (\perp \equiv \text{false}))$

If the theoretical problem of undefinedness is not easy to solve, in practice, C# has a clear solution: for each null reference throw an exception. But if a null reference exception is thrown from within a contract condition we cannot reason whether the condition was met (true) or not (false).

A solution for undefinedness is underspecification. In our first example, the second condition is underspecified by the first: its domain of values excludes the possibility of lines being null. We found the following possibilities for underspecification of contract conditions:

- `"o != null && (o.Length)"`—the left operand underspecifies the right one and all following conditions;
- `"o != null ? (o.Length) : o.Length"`—the condition underspecifies the "then" subexpression;
- `"o ?? (o.Length)"`—the left operand underspecifies the right one;
- `"o != null"`—this expression, taken as a whole condition, underspecifies all following conditions;
- `"o != null & ..."`—the left operand underspecifies all following conditions, but not the right operand;
- `"... & o != null"`—the right operand underspecifies all following conditions, but not the left operand.

Checking for possible null references is done on the Operation level. When the model is saved, each Operation element will iterate through its parameters and create a list of those which have the ReferencedType set to a Class type. These are the parameters that could possibly be null.

By checking each condition in turn, we determine if it is testing whether any of the parameters in the list is null. If the check is underspecifying subsequent conditions, then we remove the parameter from the list and continue. If one of the operands underspecifies only a subexpression, then we create a local list of parameters and remove the checked one for the underspecified subexpression.

For each parameter access inside a contract, if that parameter is still on the list, then we output a warning to the user that it might be null at that point.

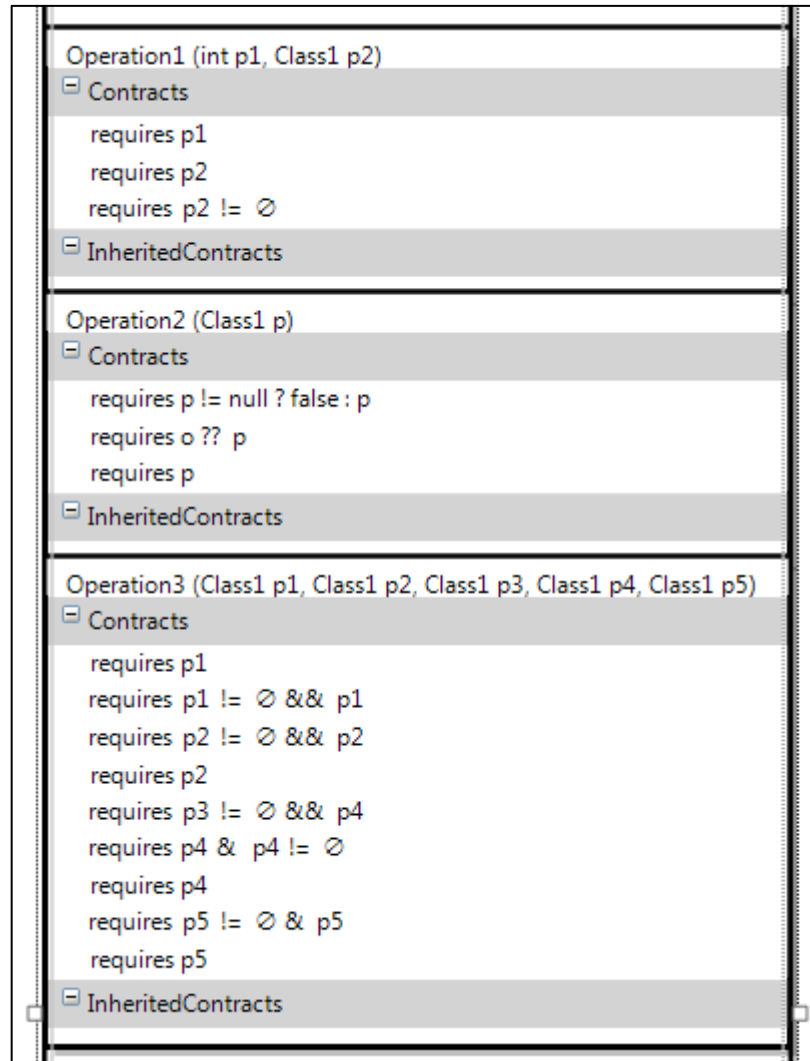


Figure 30 Undefinedness example

On Figure 30, the method *Operation1* takes a value type parameter (*p1*) and a reference type parameter (*p2*).

requires p1	does not display any warnings because a value type parameter can never be null
requires p2	displays a warning that p2 is undefined even if it is tested for nullity in the following condition

The method *Operation2* takes a reference type parameter (*p*). The parameter will show up as undefined in all three conditions.

requires p != null ? false : p	checks if “p” is null, but it only underspecifies the “then” subexpression and not the “else” subexpression so this condition will display a warning
requires o ?? p	tests if an unrelated variable “o” is null so there is a possibility that “p” will also be null at runtime
requires p	will also display a warning because none of the previous conditions underspecify this one, but only parts of themselves

Operation3 takes five reference type parameters.

requires p1	warns that “p1” could be undefined
requires p1 != null && p1	does not show any warnings because “p1” on the right side of the conditional AND is underspecified by the left side subexpression
requires p2 != null && p2	does not show any warnings for the same reason
requires p2	does not show any warning either because it is underspecified by the previous condition
requires p3 != null && p4	warns that “p4” is undefined
requires p4 & p4 != null	also warns that “p4” is undefined because it is evaluated before testing for nullity
requires p4	does not throw a warning because “p4” is tested for nullity in the previous condition (since it is a strict AND, even if the left operand evaluates to false, the right one will also be evaluated)
requires p5 != null & p5	does not show any warning; the test for nullity underspecifies both the right operator and the following contract
requires p5	does not show any warning

g. Design Patterns with Contracts

Introduction

This chapter is about design patterns with contracts. As models or designs define abstractions from a recurring problem domain, so design patterns define abstractions from models. Design patterns are especially handy when dealing with complex software architectures (31). A model or design can also be discussed easier with the use of design patterns, as they provide a common vocabulary (8).

In IDEA, identifying proven design patterns in a model are improving it (32), in a similar fashion to how identifying patterns in a software engineering process raises the quality of the process, until pattern overload starts to become a burden in improving software quality. Pattern overload is described in the publication “Using Design Patterns to Develop Reusable Object-Oriented Communication Software” by Douglas C. Schmidt (33).

Contracts make the intent behind a model explicit. Generating models from known pattern specifications is definitely feasible. Thanks to the “Design Patterns and Contracts” book by Jean-Marc Jézéquel, Michel Train and Christine Mingins (8), this section is going to demonstrate improving models by design patterns holding contracts with the example of a relatively simple Double Derivation and a more complex Abstract Factory.

Detecting a model for generating a pattern

Before applying a design pattern to a model, we can decide when exactly we provide the option for it. Even though a model does not contain any model elements - there is no existing model in the domain - we can still provide the pattern in its basic form.

This basic form of a pattern model encapsulates most of the design decisions to apply when a partial or full context is to be turned into a context-specific model that uses the principles set up by the basic pattern model. In other words, it is an empty predefined shell that has intent but needs extra work to fit into a domain context. It is not a generic model: the classes and types are not generic. At the time of this writing, we do not provide these predefined models.

Generating a new ModelElement in the diagram programmatically is considered a model transition. Model transitions themselves are required to run inside transactions. Modeling transactions require the Store object. The Store is part of the DSL API and provides creation and deletion of ModelElements, transactions, undo/redo, rules, events and access to the domain model (5).

i. Double Derived

Basic Double Derived Class

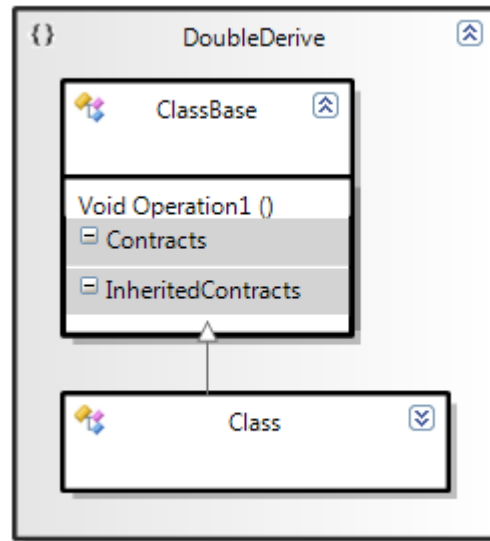


Figure 31 Diagram of a Namespace holding a basic Double Derive

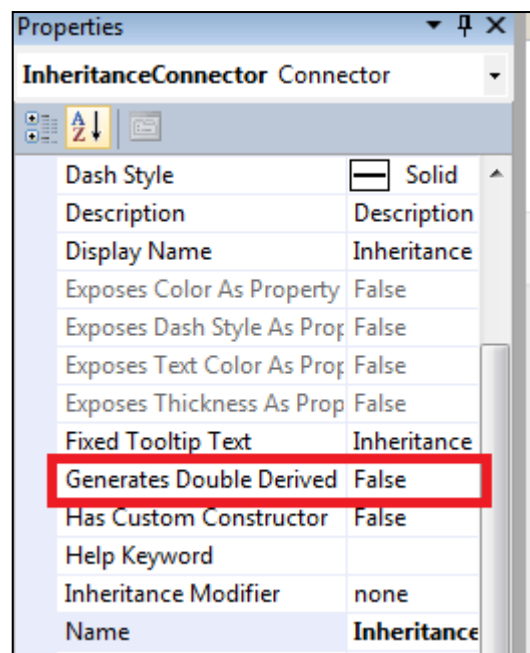


Figure 32 Some of Inheritance Connector properties in DSL Tools

Figure 31 shows the basic diagram of a double derived class. The inspiration for this model transition came from DSL Tools itself. We wanted to override some of the layout functionality of the designer in section 5.a. We realized the classes generated by the DSL engine based on the `DslDefinition` are partial. These partial classes are possible to extend in our own files outside the `GeneratedCode` folder of the DSL project. The drawback here: it is not possible to override the automatically generated methods.

In order to address this drawback, DSL Tools provides the option of generating a double derived class from ModelElements (e.g., OperationShape) (Figure 32). This way a base superclass will keep all the generated operations and it is allowed to override these operations in the derived class. Technically speaking, the base superclass is an abstract partial class and the derived class is a partial class that contains only the necessary default constructor but no other operations. After creating this partial class, we can override all the functions of the base class. An example for such an override is when the Contract Inheritance feature described in section 5.e uses an override of the GetCompartmentMappings(System.Type melType) operation on the OperationShape.

The only requirement for enabling the Create Double Derived command is to have a ClassShape with a Class selected in the diagram. This Class is to be converted into a base class.

The process of creating a Double Derived Class

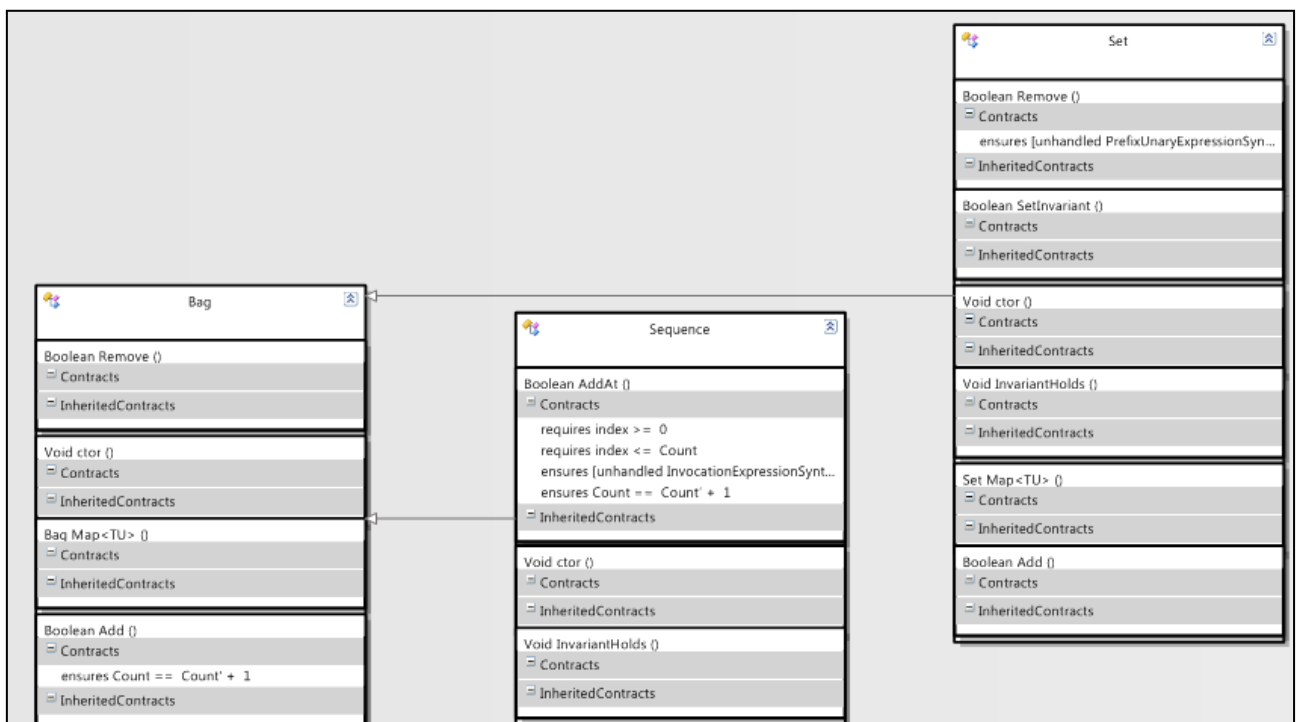


Figure 33 Library of Collections where “Set” and “Sequence” are subclasses of “Bag”

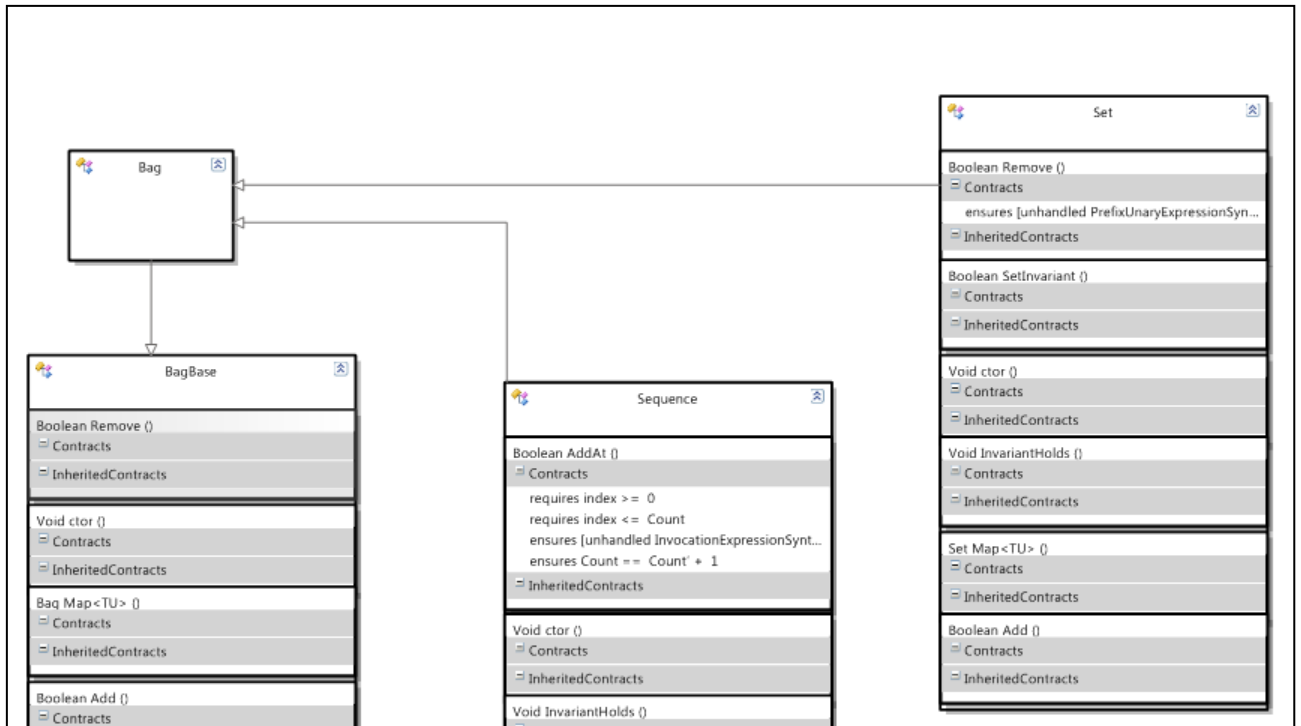


Figure 34 “Set” and “Sequence” are subclasses of “Bag”, however all the members of “Bag” have been moved to “BagBase”

An example of using our Create Double Derive command is depicted on Figure 33 and Figure 34. The “Bag” superclass is turned to “BagBase” and all its subclasses become subclasses of the derived class. This action results in the model shown on Figure 34.

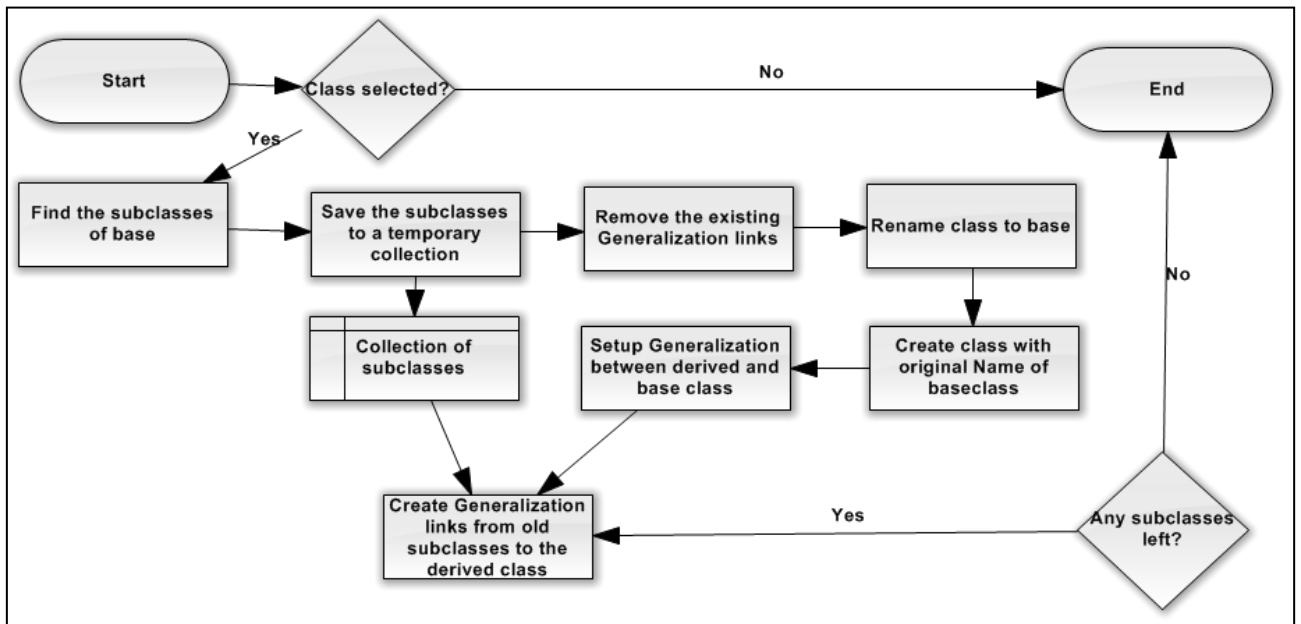


Figure 35 Double Derived Class generating algorithm

The Create Double Derive command is reachable from the designer context menu. The command is created in the same fashion as other DSL Designer context menu commands, such as the Open in Contract Editor command from chapter 5.c.

There are the following rules for generating a double derived class:

- All classes extending Class should keep pointing to Class.
- Creation of new Generalization links cannot be in the same transaction where deletion was as they would count as duplicates. A Class cannot have multiple superclasses.

Figure 35 shows the algorithm used for implementing the Double Derived functionality.

ii. Abstract Factory

Basic Factory

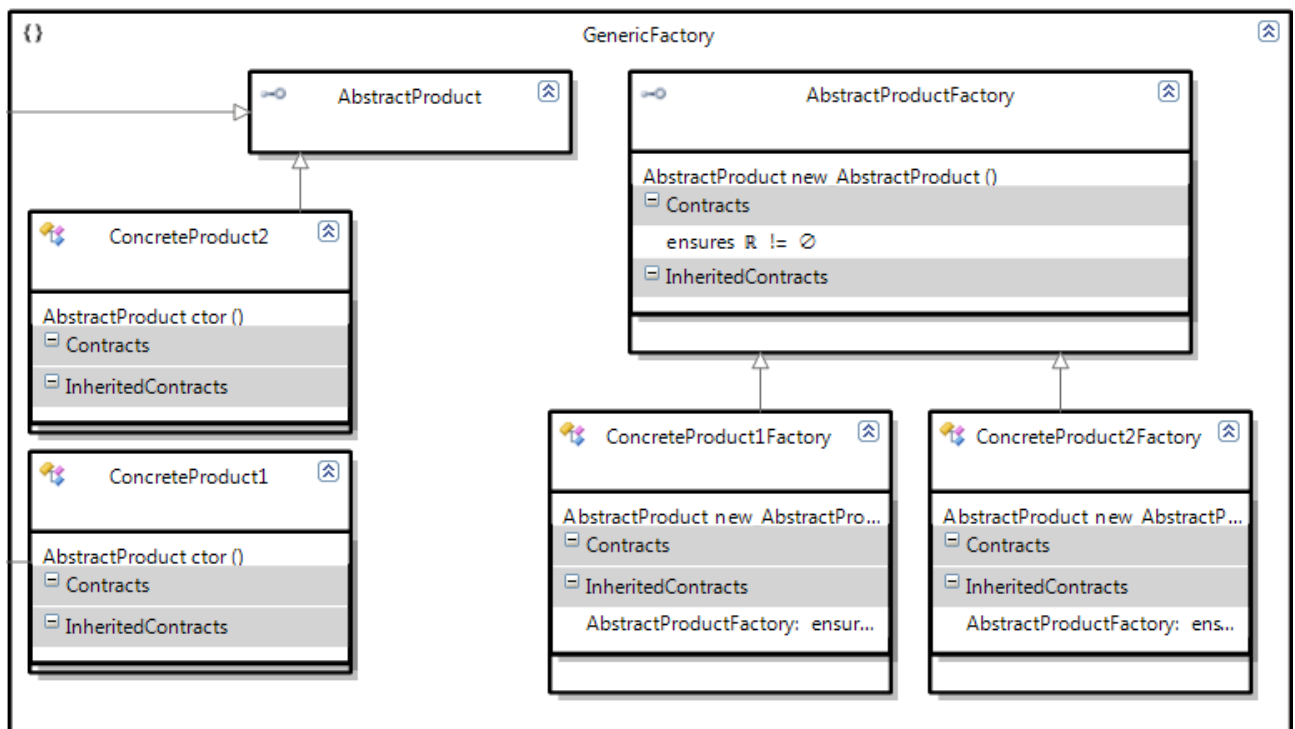


Figure 36 Diagram of a Namespace holding a basic Factory

Abstract Factory is a creational design pattern. It provides “an interface for creating families of related or dependent objects without specifying their concrete classes”. Following the descriptions of Abstract Factory and Factory Method from the book of Jézéquel, Train and Mingins Design Patterns and Contracts (8), we came up with a basic factory model.

The model is divided into a Product and a Factory part. The Product part is composed of an abstract Product (an Interface) and its implementations (as Classes with basic constructors). The Factory part has an Abstract Factory with a Factory Method to create new Products.

The predefined contract on any Factory Method is the “ensures Contract.Result() != null” Code Contract, ensuring the Factory Method does not result in a null (void object). This contract is

inherited to the factory methods in the concrete Factories implementing the abstract Factory. Figure 36 shows this basic factory model.

The only requirement for enabling the Create Factory command is to have an InterfaceShape with an Interface selected in the diagram. This Interface will be the product Interface.

The process of creating a factory

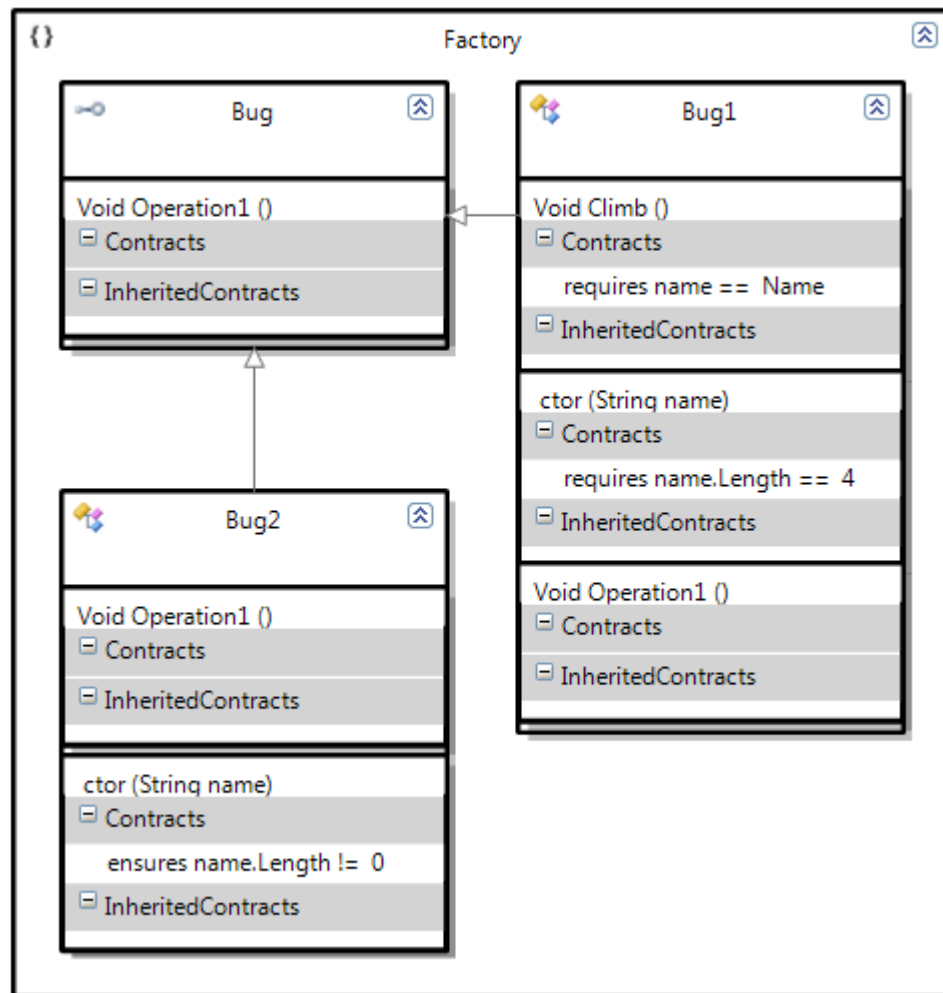


Figure 37 Interface “Bug” with two implementations inheriting its “Operation1” Operation. “Bug1” and “Bug2” share an overloaded constructor with the SignatureParameter “name”

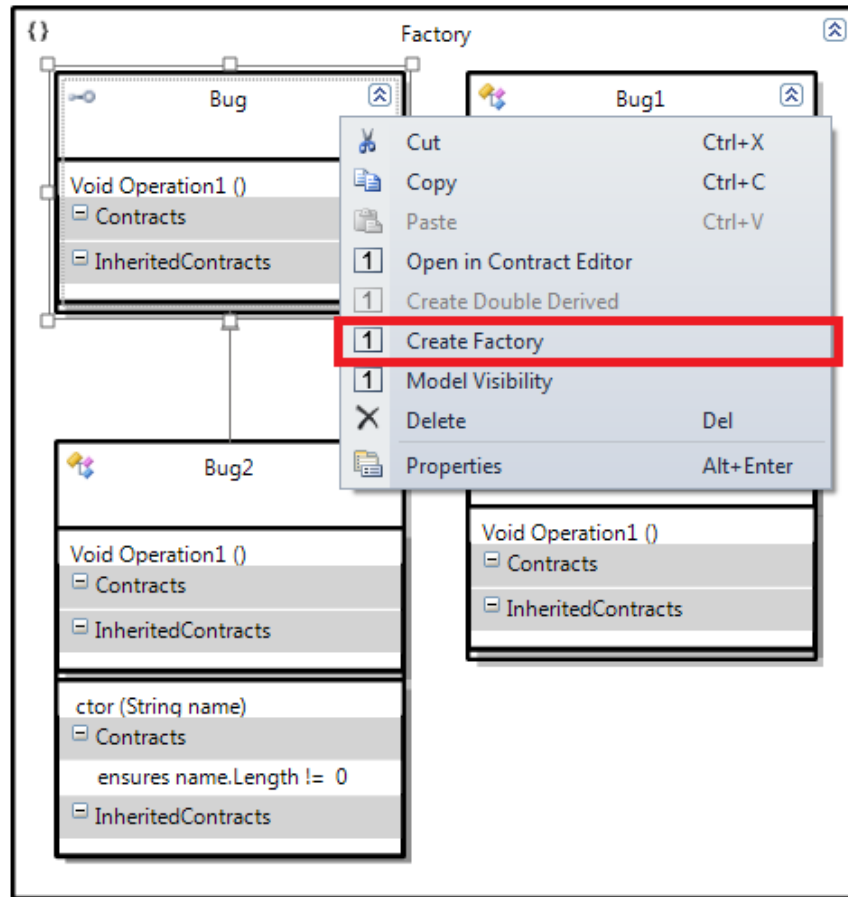


Figure 38 Create Factory designer context menu command available on selecting an Interface

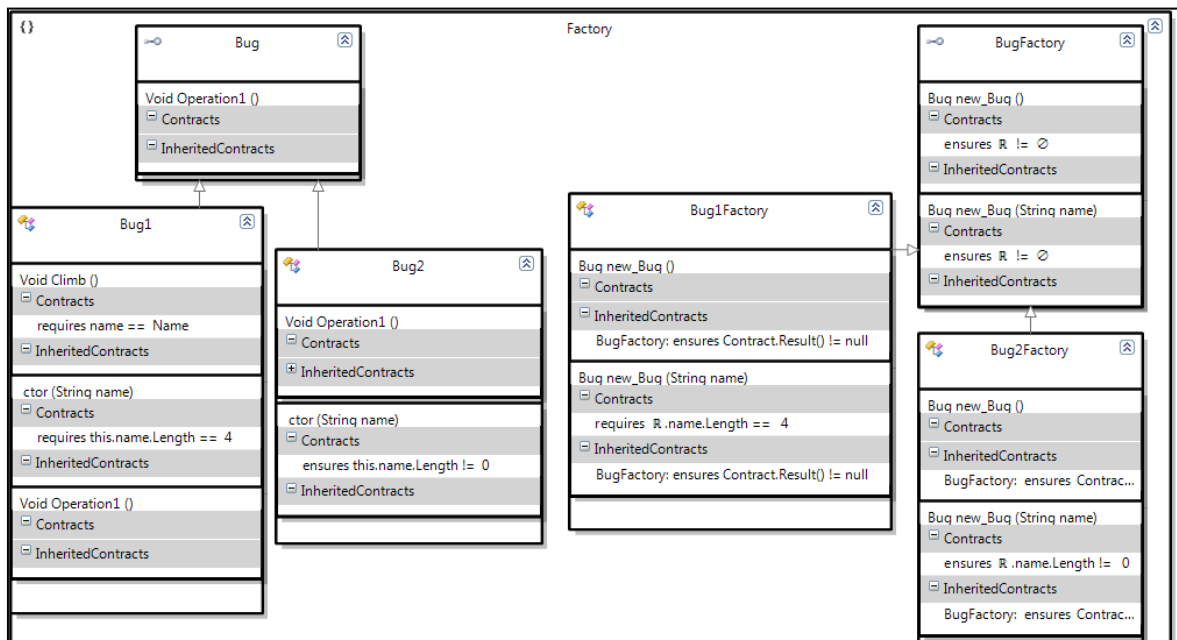


Figure 39 Result of the Create Factory command on “Bug” Interface

Code Contracts are not inherited on overloads of an Operation unless these overloads of the default factory method are present on the Abstract Factory (reason described in section 5.d about Contract Inheritance).

The Create Factory command is reachable from the designer context menu, just as Create Double Derive. It is a requirement to have a model ready in the designer which is similar to the “Bug” and its implementations in Figure 37. Similar in this case means there is an Interface and a Class implementing it. The user of our extension has to carry out the step shown on Figure 38. This action results in the model shown on Figure 39.

Implementation details

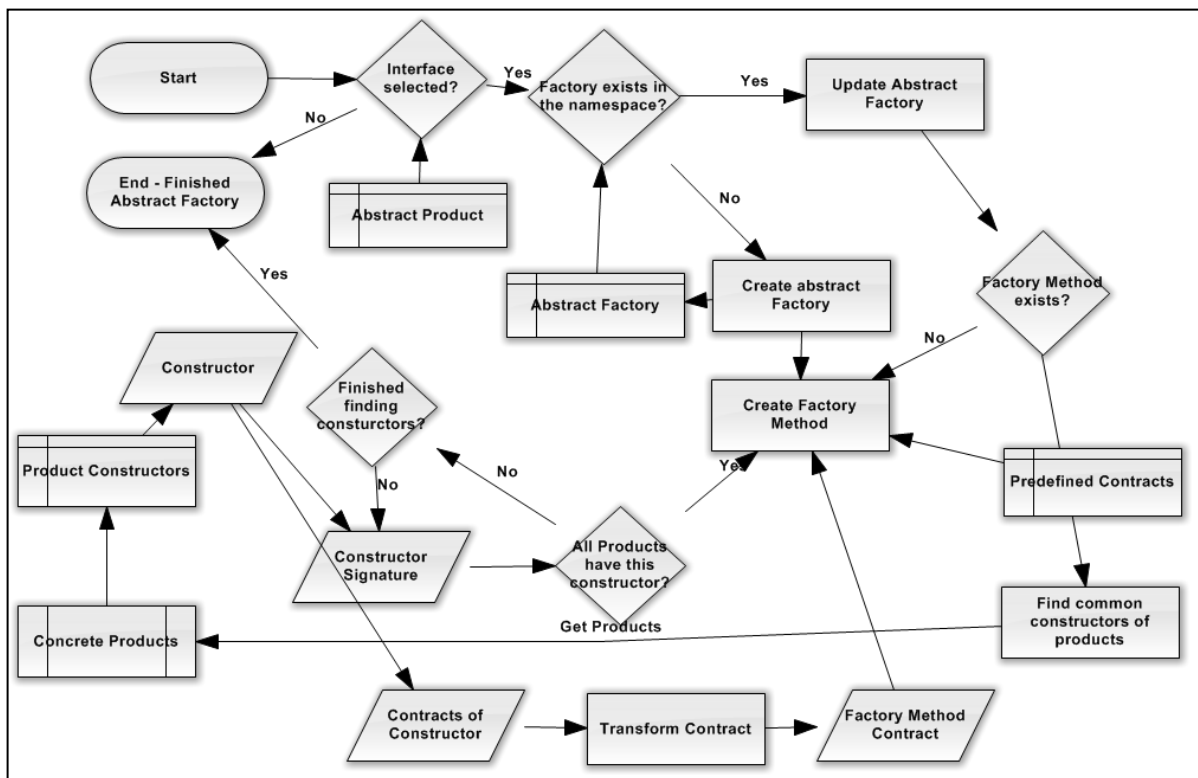


Figure 40 Abstract Factory detection and generation in the model

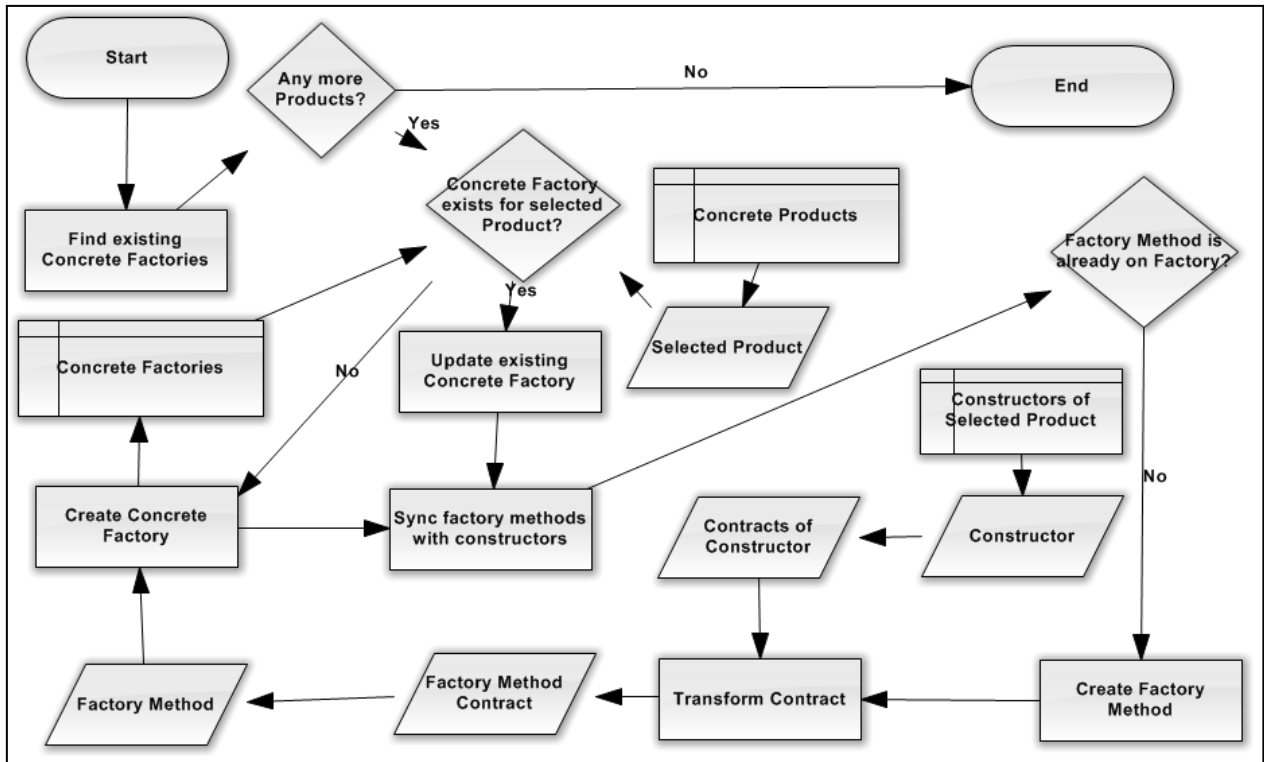


Figure 41 Generating concrete Factories in the model

There are two major stages of generating an abstract factory: generating the Abstract Factory Interface and generating the Concrete Factory Classes. The figures Figure 40 and Figure 41 show the major steps in these two stages as flowcharts. The following steps are not included on Figure 40:

- collection of all selected ModelElements
- “Factory” postfix is added to the all Product names (abstract or concrete)
- any Factory Method will be constrained by the predefined contract

The following steps are not included on Figure 41:

- Factory Methods on every Concrete Factory inherit the predefined contract from the corresponding Factory Method of the Abstract Factory. This is important as Factory Methods can be overloaded too, if the constructors of the Concrete Products were overloaded.
- In case a Concrete Product has one or more constructors that are not common with the other Concrete Products, the corresponding Factory Method will not be added on the Abstract Factory, only on the corresponding Concrete Factory. This means the Factory Method does not inherit the predefined contract, but does get its own predefined contract

Additionally, all Factory Methods get the “new_” prefix followed by the name of the Abstract Product, e.g., “new Bug”.

Patterns into existing code

Generating patterns is especially useful when it serves to improve existing code. The next section “Parsing code for building architecture” (section 5.h) shows how we map existing C# code to our model presented in section 5.a.

h. Parsing code for building an architecture

i. Various methods for parsing VS2010 solutions for files

One of the planned features of our extension was to generate models from existing C# code. For this, we would have to process all the projects in the active solution to retrieve the code model and transform it to fit the DSL model.

In our first attempt we created a Visual Studio Add-in. The entry point in an Add-in program is the **OnConnection** event handler where a DTE object is available as a parameter. From the **DTE.Solution** we would access all open projects in the current instance of the environment.

Starting small, we limited functionality only to C# projects. For each of these we traversed all its project items looking for items that represented physical files and for these, we would process the **FileCodeModel** through a collection of **CodeElements**.

Since Microsoft documentation usually follows one of two patterns - either next to none or so much that you find yourself lost in it - we spent a lot of time in a discovery process which culminated in the following paragraph: “The code model allows automation clients to avoid implementing a parser for Visual Studio languages in order to discover the high-level definitions in a project, such as classes, interfaces, structures, methods, properties, and so on. The Visual Studio core code model avoids language-specific areas of code, so it does not, for example, provide an object model for statements in functions or give full details on parameters.”²³

The bottom line was that we were able to create the model, but the contracts existing in the method bodies were only available as plain text. At that point, we had no knowledge of Roslyn. All C# parsers that we found were either expensive commercial products or they were not covering the latest versions of the language.

The breakthrough came with the realization that the Code Contracts Editor Extensions are somehow able to detect and analyze contracts in the active solution. Upon examining the contents of the extensions package we discovered the CCI assemblies and shifted our focus on the CCI project.

ii. Building the AST

Throughout the entire extension, the program we analyze is represented on several levels of abstraction. We have C# source code, CIL code in assemblies, the CCI code model and AST mapping the CIL, the XML file containing our model and the model tree defined through the DSL tools. Due to the fact that the APIs for working with CCI and DSL trees are very different in structure and inner state, we chose to create one more abstraction level. This abstraction level is a code model in which elements would hold a minimal set of properties common to the CCI and DSL models. The code model serves as a middle layer: provides an easy transition between CCI and DSL

²³ <http://msdn.microsoft.com/en-us/library/ms228763.aspx>

models. Due to a nomenclature confusion, we called this code model an “abstract syntax tree” (the Ast class in our code) and placed it in the **Company.SolutionArchitectureLanguage.AST** namespace, even though it has the function of a code model. Since this is how it currently appears in our code, it will be called simply “the Ast” further in this report, as opposed to “the CCI model” and “the DSL model”.

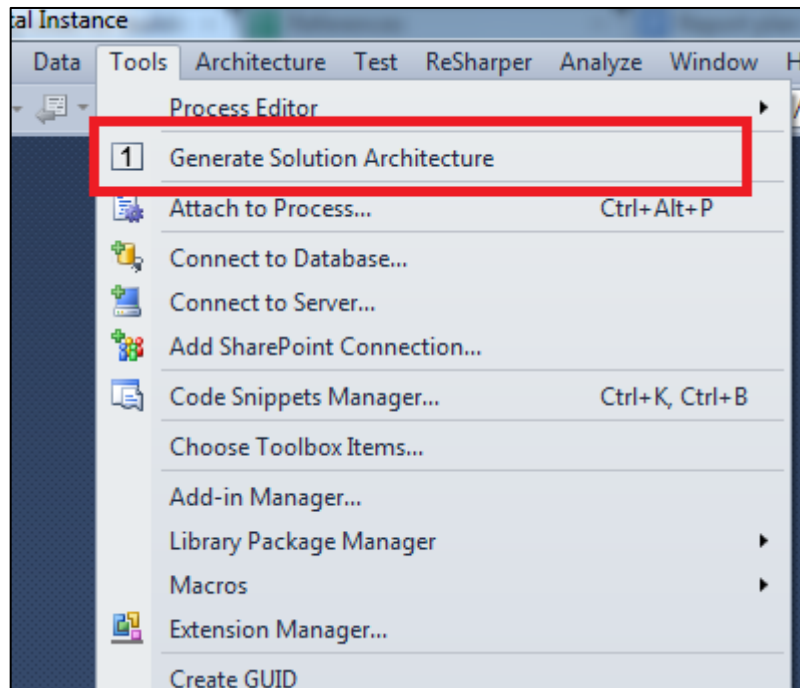


Figure 42 Visual Studio Tools menu

When using our extension, one can generate a model from the active solution by selecting the Tools menu and clicking the “Generate Solution Architecture” item (Figure 42). First, the command builds the active solution to ensure that the assemblies are up to date and ready to be read with CCI. Then it proceeds to building the tree.

From our initial experience we found that building models from large solutions is a lengthy process that freezes the Visual Studio UI if it runs in the same thread. However, when the model is done we need to create a project item on the UI thread; otherwise it is not accessible to the user. The following steps describe our solution.

Visual Studio UI thread: MenuItemCallback	<ol style="list-style-type: none"> 1. build the solution 2. instantiate an Ast object 3. setup BackgroundWorker 4. run it asynchronously (pass the Ast object as parameter)
BackgroundWorker thread: BackgroundWorker DoWork	<ol style="list-style-type: none"> 5. get the DTE object from the Package 6. get the active solution and iterate through all projects 7. for each C# project get the active configuration and the output path 8. detect the main assembly (dll or exe) and the contract reference assembly 9. load each assembly in a CCI CodeContractAwareHostEnvironment 10. use a CCI Traverser to traverse the model and transpose it in the Ast object 11. return control to the UI thread
Visual Studio UI thread: BackgroundWorker RunWorkerCompleted	<ol style="list-style-type: none"> 12. serialize the Ast object in a temporary file under the Local Application Data folder 13. launch a modal dialog window for the user to pick the location where the project item is to be saved 14. create the item from the temporary file

Step 1:

Building the solution is done through the **DTE** object. The build is done on the Visual Studio UI thread and the process waits for the build to finish before it continues.

```

1. public static void Build()
2. {
3.     var dte = Package.GetService(typeof(DTE)) as DTE;
4.     if (dte != null)
5.         a. dte.Solution.SolutionBuild.Build(true);
6. }

```

Step 2:

The minimalist model tree we created is mimicking the model tree created in DSL tools.

Most classes in the minimalist model inherit from the **IAstElement** interface. This interface defines an **Id** property (a globally unique identifier that identifies model elements in the DSL) and a **Moniker** (a form of referencing such elements).

In the DSL, monikers are used in reference relationships (e.g. when a class implements an interface, it is the interface moniker that points to it). Every element has a moniker key that can be either the id or a name property, if the element name is unique among siblings. For example, namespace and class identifiers are unique element names, but a class can contain multiple

operation overloads with identical identifiers. In this case, an operation's moniker would be "model_guid\namespace1_name\namespace2_name\class_name\operation_guid".

In the minimalist model we have set monikers for elements that are used several times during the model building process: namespaces, types and class members. Signature parameters, generic parameters and contacts are only encountered once (when they are added to the operation) and do not need a moniker. When traversing the CCI model, we might end up visiting the same overload of an operation more than once (i.e. once in the main assembly and once more in the contracts reference assembly). In this case, we must not add the operation again, but find the original and just insert the contracts. However, the id of the operation is set on the first occurrence and there is no way of retrieving it from the CCI model. Because of this, the operation moniker for the minimalist model is formed of the operation signature (identifier and parameter types). This is the only case where a moniker is not identical between the two models (minimalist and DSL).

The rest of the elements are fairly simple.

The **AstSolutionArchitectureModel** class represents the root of the model tree. It contains a collection of **AstNamespace** objects.

The *NamespaceMember* is an abstract class in the DSL and an interface (**IAstNamespaceMember**) in the minimalist model. It contains a definition for the member Name property which is to be unique among siblings.

AstNamespace members can be other **AstNamespace** instances or **AstTypes**.

The *Type* abstract class of the DSL (and the **IAstType** interface, respectively) does not bring additional functionality.

The **AstStructure**, **AstDelegate** and **AstEnumeration** elements have been added, but we have not focused on their developments. They do not have child elements and do not reflect the full features of their C# counterparts.

The **AstInterface** element has a collection of references to other **AstInterfaces** it implements and a collection of **AstAttributes** and **AstOperations** it defines. Even though in C# interface operations do not have bodies and Code Contracts are defined as method calls, CCI is able to extract contracts for interface members. The models permit interface operations to contain contract elements.

The **AstClass** element is similar to the **AstInterface**: it also references implemented **AstInterfaces** and has a collection of **AstAttributes** and **AstOperations**. Additionally, it can reference at most one other **AstClass** as a superclass and can contain inner **AstTypes**.

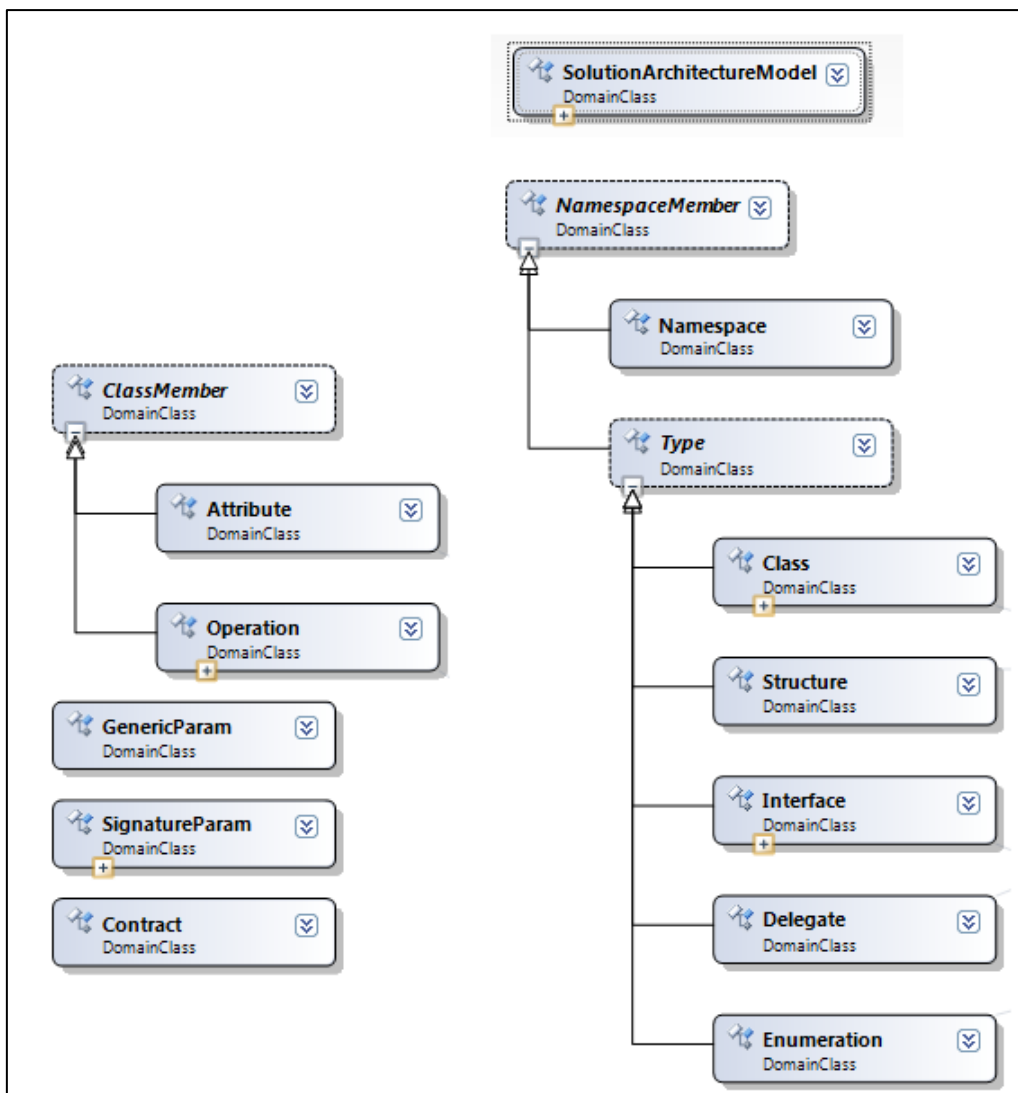
The **AstAttribute** maps the Attribute model element. It has an **Identifier** and a **Type** property, both text strings.

Beside the **Identifier** and the **Type** properties, the **AstOperation** contains collections of signature parameters, generic parameters and contracts.

The **AstSignatureParam** has an **Identifier** property and a **Type** property. The *SignatureParam* in the DSL model has an extra *ReferencedType* relationship to a type that exists in the model. The reasoning behind this is that we started with the simple string Type property which allowed naming the type. We needed this because parameter types could come from unmodeled assemblies like System, in which case we would simply name them. Later on, we realized that it could be useful to know whether the type of the parameter is a reference type or a value type so we added the *ReferencedType* in the DSL. The user can only set the Type or the *ReferenceType* of a *SignatureParam* object.

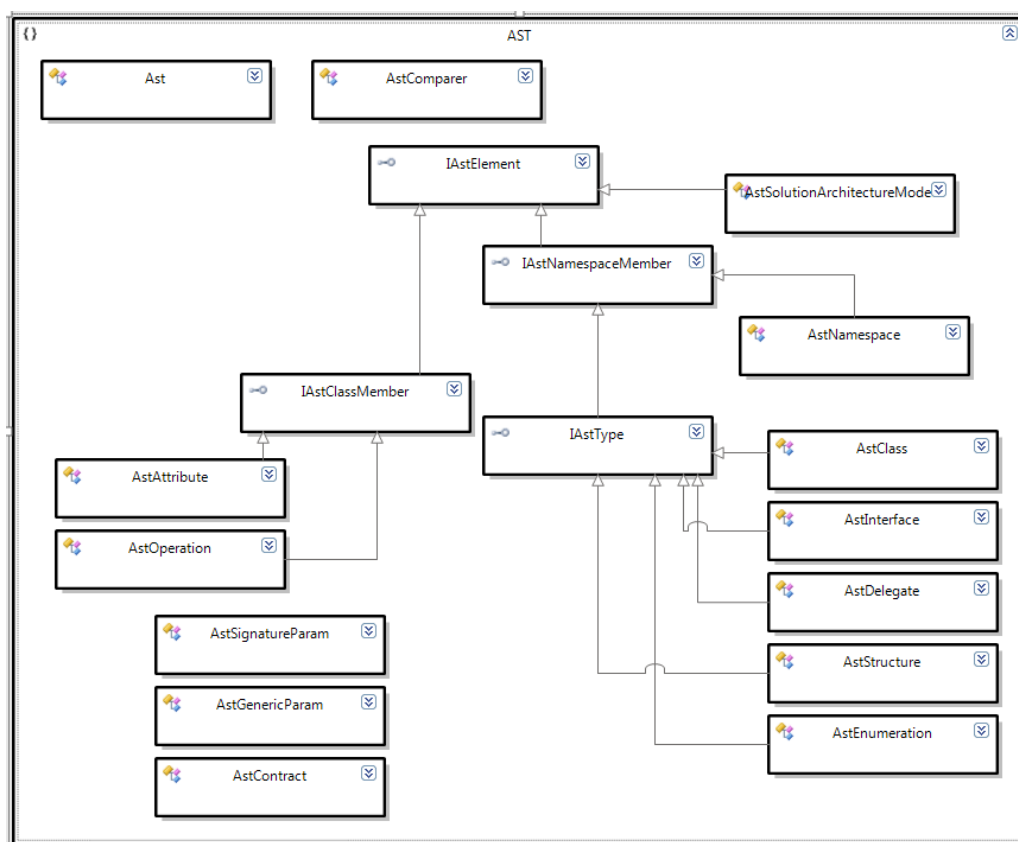
The **AstGenericParam** only has a **Name** property (called *Identifier* in the DSL). It is the name of the type which is referenced.

The **AstContracts** element has an **Expression** property of type string. This property is the contracts condition as it is extracted from the CIL code by CCI. The expression is not strong typed within the model since it may refer to variables of types defined in other assemblies. This also means that, in the current form, the model only supports the basic forms of contracts which only take as parameter the condition.



The **Ast** object is a wrapper that contains the **AstSolutionArchitectureModel1**. It also contains a dictionary that stores references to all elements in the model based on their monikers. The reason behind this is that whenever we need to access an element and we can compute its moniker, we don't have to traverse the tree, but we have a direct reference to it through the dictionary.

To enable this, all elements must be added through the **Ast** object. Whenever an element is added, its parent's moniker is also passed as a parameter. If the parent element is found (by moniker) in the dictionary the new element is added both as a subelement and in the dictionary to be referenced later. The only elements that need not be referenced after they are added are the **AstSignatureParam**, the **AstGenericParam** and the **AstContract**, hence they do not have monikers and are not inserted in the dictionary.



Steps 3 and 4:

Setting up and running the **BackgroundWorker** is done in a trivial manner.

```

1. var backgroundWorker = new System.ComponentModel.BackgroundWorker();
2. backgroundWorker.DoWork += BackgroundWorkerDoWork;
3. backgroundWorker.RunWorkerCompleted += BackgroundWorkerRunWorkerCompleted;
4. backgroundWorker.RunWorkerAsync(ast);

```

Steps 5 through 8:

Retrieving the list of projects under the active solution is done recursively since they may be nested under solution folders. We also filter them based on their kind, keeping only C# projects, even though CCI is able to map the architecture of any .NET assembly. However, we decided to start with a familiar language.

For each project we retrieved the active configuration, since different configuration produce different output folders. In the Configuration object the path to the output folder can have various patterns:

1. "\\server\folder";
2. "drive:\folder";
3. "..\..\folder";
4. "folder".

After identifying which one is used, we combine the absolute path with possible assembly file names: "\\AssemblyName.exe", "\\AssemblyName.dll" or "\\CodeContracts\\Assemblyname.Contracts.dll".

Step 9:

CCI is able to load assemblies from a variety of sources: hard drive, memory, online locations or even assemblies embedded in Microsoft Office Word documents. Hosts are abstractions of these sources that provide functionality defined by the **MetadataReaderHost** abstract class. One of the CCI samples provides a **CodeContractAwareHostEnvironment**, a host capable to extract contracts for any code model element. Each assembly discovered in the previous step is loaded through this host in a CCI Module object.

Steps 10 and 11:

We retrieve the model elements from the Module object with a custom Traverser that inherits from the **BaseMetadataTraverser** class. The base class contains overloads of the Visit method for each element type encountered during the traversal (e.g. **IAssembly**, **ICustomModifier**, **IEventDefinition**, **IGenericParameter**, **IMethodBody** or **ITypeDefinition**). The traversal of the tree is handled by these virtual methods. We only needed to override the important ones and ensure the base methods were called for the traversal of the child elements.

The elements we were interested to visit are: **INamespaceDefinition**, **ITypeDefinition**, **IMethodDefinition** and **IFieldDefinition**.

This step has two phases. In the first phase we traverse the model and insert all important elements in the **Ast** object. However, there is a problem when an element is referencing another. For example, when we visit a class which implements an interface, that interface might have not been visited so it might not exist in the **Ast**. If the interface is defined in another assembly, it might never be visited, since we only process assemblies from the current solution. For this reason, the list of interfaces an **AstClass** is implementing contains only the monikers of the

interfaces (we can compute the moniker at any time). If the interface is visited at a later point, then the reference is fine. If the interface never gets visited, the moniker will refer to an object that does not exist in the **Ast**.

The second phase is a validation. After we have built the minimalist model tree from all available assemblies, we traverse it and remove all monikers that do not exist in the element dictionary.

After the validation, we return control to the Visual Studio UI thread.

Steps 12, 13, 14:

There are several ways of creating a Visual Studio project item: either using an item template (in which case it will have the default content) or from a file on the hard drive (in which case we can control the content).

To accomplish this, all **Ast** elements have a **Serialize** method which creates an XML file that can be opened with the DSL designer. We save this file in a temporary file under the Local Application Data folder, and then create a project item as a copy of that file. The location where the project item is saved in the solution tree is chosen by the user through a modal dialog window.

Even though the DSL designer is using two XML files (one for the model contents and another for serializing the designer layout) we only need to create the model XML; the layout XML is created automatically when the model is loaded for the first time in the designer.

Generating the layout XML in the future would bring a serious benefit. As we have created our own layout algorithm we can apply it either upon a user command or on the load event of the designer. Setting it on the load even means that layout changes done by the user (by moving the shapes) will be reset the next time the designer loads. By applying our layout algorithm when the file is created we can ensure that the shapes will be laid out properly when the designer opens, but user changes will also be preserved.

iii. Code-Model Comparison

Code-Model comparison can be launched by:

- selecting a model project item in the Visual Studio Solution Explorer;
- selecting the “Verify Against Code” menu command in the project item’s context menu.

By launching this command from the Solution Explorer means that the model designer does not have to be loaded. The developer can simply create the model first, close the designer and move on to coding, comparing the model to his code with just two clicks when needed.

When the command is launched two **Ast** objects are created: one from the actual code (as described in the previous section) and one from the XML file of the selected model item.

To retrieve the XML, we use the **EnvDTE** library to find the selected item in the Solution Explorer, check if it is the correct type of item and get the path to its physical file. To create the **Ast** object, we use the **Parse** method which is the opposite of the **Serialize** method.

After both **Asst** objects are created, we traverse them in parallel identifying elements that they do not have in common. Each of these elements is reported to the user specifying whether it is located in the code or in the model and displaying the full path to the element.

ERROR:	Code	Operation	/9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ ParseTest/Class1/Test does not exist in model!
ERROR:	Code	Operation	/9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ ParseTest/Class1/ctor does not exist in model!
ERROR: Code Interface /9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ ParseTest/ITest does not exist in model!			
ERROR:	Code	Class	/9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ ParseTest/ContractsForITest does not exist in model!
ERROR: Model Class /9541b7ab-fd13-4ff4-8f3b-ecf00f7815fb/ ParseTest/TestClass does not exist in code!			
ERROR:	Code	Namespace	/9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ ShorthandedContracts does not exist in model!
ERROR: Code Namespace /9de58c3b-26aa-4c0b-b785-42ee96cb75f7/ System does not exist in model!			
ERROR:	Model	Namespace	/9541b7ab-fd13-4ff4-8f3b-ecf00f7815fb/ UniversityApplication does not exist in code!

6. Discussion about goals in the project base

The previous section was about how we made use of our findings in the major focus areas of this project: Design by Contract, Domain-Specific Languages, Model-Driven Development. However, we would also like to discuss the impediments and obstacles we encountered on the way.

1. The tool should help developers model their application in a usable way.

Completion of this goal is subjective. There is no formal way of verifying it. This goal is a statement about the quality of our research product. Further research specific in reviewing model-driven development tools would be required to assess this quality. The research could be based on feedback from the model-driven developer community, just like developer students answered questions about their modeling experience in “Building tools for model driven development comparing Microsoft DSL tools and eclipse modeling plug-ins” (21). Another research was carried out in the publication called “Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling (In the context of the Model-Driven Development)” being one of the examples. In this publication DSL Tools were compared to the Eclipse modeling extensions on the following criteria (34):

- Conformity: How well is each domain concept mapped to a language construct? The language constructs must correspond to important domain concepts.
- Orthogonality: Does each language construct map only a single distinct domain concept?
- Supportability: What is the coverage of the modeling tool in terms of features for model management and model transformation?
- Integrability: How well does the modeling tool integrate to the development environment?
- Longevity: How long does the tool support last?
- Simplicity: How simple the DSL is in comparison to the domain concepts it tries to model?
- Quality: What mechanisms does the DSL provide that improve the reliability, security, and safety of an application?
- Scalability: How easy it is to model large descriptions or specifications by the DSL constructs?
- Usability: How expressive and understandable are the constructs in the DSL? This evaluation criterion strongly depends on the combined evaluation of the above criteria.

Unfortunately, the usability features for example in AMOC’s Contract Editor need more implementation work: validation, editing parts of the signature separately, and changing the ModelElement’s Name. These are only a few from the countless features missing that prevent AMOC from being published.

2. It should provide an easy transition from model to code and from code to model.

Our solution is not complete. The "code to model" translation is described in section 5.h. Our DSL is a mixture of UML and C# language constructs and therefore it can be considered both a modeling language and a programming language. There are also vast amounts of C# features we do not map into our model. Examples are referencing types from other assemblies, full trees of Code Contract specifications, modifiers and attributes (as attributes in C# that are accessed by

reflection²⁴). Admittedly, we only map a handful of C# features in order to demonstrate a Design by Contract proof-of-concept.

The "model to code" translation is a matter of code generation. The abstract syntax tree of high-level language constructs can for example be modified using the features of the Roslyn compiler API. However, writing code to existing code files requires completeness from the model, as the existing code files can use any C# feature. "Code to model" translation should be complete before code is generated from our model.

3. Display inconsistencies as errors or warnings upon saving a model.

The model should always be well-formed but it allows writing not well-typed parts. The not well-typed parts will not be checked against source code. This should allow users to model without worrying about updating the code the same time.

A "verify against code" comparison is described in section 5.h and needs refining. It provides useful error messages to allow users to check the validity of the model.

The well-formedness of the model (the relationships between model elements) is checked by employing DSL Tools mechanisms. However, the type of various model elements is stored as simple strings and it is only checked by comparison with the types of those elements on the code side.

4. It should make use of Visual Studio Extensibility such as Domain Specific Language Tools and Add-ins.

We use features of both extensibility frameworks (DSL Tools and Visual Studio Extensibility). The goal is completed as we have learned a lot from both. We refer to our findings in sections 4 and 5. Obstacles in this research process have been numerous.

The frameworks are rich in features. The APIs provided are pieces of work from many years. On the other hand, we often consider it non-trivial to achieve concrete goals. The documentation for DSL Tools is not yet complete, especially considering layout mechanisms of the class diagram designer. We did not find indications of certain required events on classes for graphical elements that are standard in other .Net based graphical frameworks as WPF or Windows Forms. An example for such an event is the Load event which occurs when the control contains all the child elements but is not shown yet²⁵.

5. It should help .Net developers make use of .Net Code Contracts in a simplistic way.

Completion of this goal is subjective. There is no formal way of verifying it. We refer to the research suggestions for point 1 in this listing of goals.

6. Working in both synchronized and asynchronous modes (between model and code).

The extension works in asynchronous mode: code and model are updated individually and compared to check validity. Synchronization of code and model must be done manually. The

²⁴ <http://msdn.microsoft.com/en-us/library/ms173183.aspx>

²⁵ <http://msdn.microsoft.com/en-us/library/system.windows.forms.usercontrol.onload.aspx>

benefit of asynchronous update is that incomplete models can be persisted. The development of an automatically synchronized mode is subject for further work. A requirement for this is finding out if any changes have occurred in the code when a code file is saved to the file system. Ideally, only the changes should be updated into the model avoiding the regeneration of every construct. The synchronization of invalid or incomplete code to model requires parsing C# code as text and not from built assemblies. Assemblies are not built if the code has errors, so our current CCI parsing will not create a model. We could use Roslyn to create incomplete models from the incomplete code however, as Roslyn is able to parse and validate code constructs as text.

When synchronizing changes that occur in the model (model to code), we need to know where the changes originate from: capturing events in the DSL designer, capturing events changes from the Visual Studio XML Editor for the XML file (open if the DSL designer is closed), capturing a change in the Visual Studio project when the XML version of the model diagram is edited by an external XML editor.

Synchronizing could also happen by skipping the serialization to the textual (XML) format and generating only C# code. However, in this case the graphical model would have to follow precisely the C# language as model information would not be persisted anywhere else than the C# code files. This brings the inconvenience of forcing complete models. Diagrams would only persist which code files to show. Class Diagrams in Visual Studio provide this functionality.

7. Research ways for creating a modeling tool, taking the concerns of the current DSL developers into consideration.

Such a research could be based upon reviews of modeling tools (as briefly mentioned in point 1). Originally, we were developing the extension to include enough features for public testing and reviewing. In the end, we have developed a modeling tool for Visual Studio while taking a look at the current modeling tools for different platforms (Eiffel and Eclipse). However, we did not manage to reach a state where we felt confident publishing the solution.

8. Representing domain specific languages in visual and textual format.

Visual (diagram) and textual (XML) formats are supported automatically by DSL Tools. We also use a Contract Editor tool window to aid the designer. Concrete features of this tool window are described in section 5.c.

9. Visualization of a contract in a graphical invariant designer.

Textual representation of the contracts is using short forms of contract expressions. The use of graphical symbols as in graphical programming is not part of the scope of the current thesis. As there can be many contracts on many methods inside many classes or interface, scalability of the model when using graphical representations for every small detail would be a challenging task. Finding standards for the graphical representations could form the subject of another research subject.

10. Error messages stating the violation of a constraint in a friendly language (connecting contracts to user-understandable error messages).

This goal came from looking at Eclipse Modeling plugins being used for creating DSLs that are not general purpose. Our DSL is general purpose since it models features of a general purpose modeling language and a general purpose programming language. We can only put those constraints into a DSL that come from the domain it solves. As our domain is general purpose programming, we do not limit the model design process (creating a class diagram) by Code Contracts. However, we do support using Code Contracts as parts of our model. Validation of constraints happens based on this model. We do not currently modify how Code Contract warnings and error messages are displayed. An addition to these error messages is our warning for undefinedness.

11. Check contract expression style, undefinedness and perform other checks that the C# compiler and the Code Contracts Static Checked are not able to.

Suggest alternate ways of writing the contracts in order to improve their effect.

The fundamentals in this area are implemented, but more examples are needed for completeness. This goal has been accomplished in connection with undefinedness and the contract expression styles in section 5.e. We provide more meaningful contract exceptions on expressions that may evaluate to null resulting in a null reference exception and suggest ways of rewriting the expression in a more useful form.

12. Automate the suggestion of contract templates for implementation of design patterns.

Automatically generate the core structure of design patterns in the model, based on existing model classes.

We generate the double derived class and abstract factory patterns (section 5.f) as a proof-of-concept. More creational, concurrency, behavioral and structural pattern examples could be implemented following specifications such as the ones in the “Design Patterns and Contracts” book (8).

7. Conclusions

AMOC did not reach the public availability stage in this project. Open-ended questions like “How would AMOC fit the needs of current developers?” are not yet answered. However, our research question “is it possible to include Design by Contract in Visual Studio modeling?” is answered, as AMOC demonstrates a proof-of-concept solution for visualizing models with Code Contracts. The developer can create a class diagram from scratch or generate one from an existing Visual Studio solution with C# code files. Using CCI to parse .NET assemblies would not require for the code files to be C#-exclusive, but parsing Code Contracts by Roslyn is language specific.

AMOC is not a general solution and, even as a very specialized modeling tool, it needs more tailoring. An example is the class diagram: class diagram elements should explicitly be named after UML or C# models, not a mixture.

As bottom line, AMOC is an incomplete extension, but we have learned a lot by creating it. The Perspectives chapter briefly mentions a few ideas for building on this knowledge as future work.

8. Perspectives

Features from our solution are currently implemented only as a proof-of-concept. We imagine many directions of improving them, some more farfetched than others. Such a direction is reaching for completeness. In the Discussion section we discuss the option of replacing CCI parsing with Roslyn. As Roslyn might become the next official .NET compiler, AMOC would incorporate modeling more C# features as the Roslyn parser evolves to reach full C# coverage.

Our implementation code was written with defensive programming. Therefore, another direction for improving the solution is striving for software quality. A modeling tool designed to advocate Design by Contract should itself be following the Design by Contract way. Converting our defensive code to Code Contracts and creating new functionality following Design by Contract principles are clear possibilities for raising the safety and correctness of our tool.

We were inspired by several EiffelStudio features. However, the official IDE for the Eiffel language is a pool of interesting ideas, such as showing a model with or without ancestors, inheritance, reference links, referenced models (models from imported libraries). EiffelStudio is showing source code and class diagram structure from the Eiffel language's own libraries. Using Roslyn we would be able to parse the referenced libraries and generate a model resembling the Diagram tools functionality from EiffelStudio.

In the future we will update the designer to fully support the contract inheritance. The correct way of doing this is to reflect the runtime checker behavior: go up the inheritance tree of a method (starting with the local contracts) and collect contracts from the superclass, interface, superinterface and so on. Unlike the Editor Extension adornments, we will show the actual location where each contract is defined. Another improvement that we want to bring is to highlight or focus on the parent superclass or interface when an inherited contract is selected. Another problem when updating the designer to support full inheritance will be displaying the correct warnings. As we have shown, warning messages are inconsistent between usage scenarios so we will have to further research the matter and, possibly, ask for clarification from the Code Contracts developer team.

A model in AMOC contains model elements. If such a collection of model elements is a reusable model with an intent (like design patterns), the opportunity opens for reusing parts of the existing model as snapshots. Such a snapshot would only reference a specific subset of model elements (for example a namespace or a class) from a model. While showing a single snapshot in the diagram is a scalability feature, copying such snapshots with specifications into a diagram is creating the shell of specification-based applications. Snapshots could also come from (and extend) a repository of design patterns. If the extent of snapshots is not limited, domain-specific software architecture templates with model correctness in mind could be reused when developing applications.

As mentioned in section 6, publishing the project is the next step for receiving useful feedback from the developer community. Feedback would shape which areas of the tool should receive focus.

9. Bibliography

1. **Meyer, Bertrand.** *Design by Contract*. s.l. : Interactive Software Engineering Inc., 1986. Technical Report TR-EI-12/CO.
2. **Mandrioli, Dino and Meyer, Bertrand.** *Design by Contract. Advances in Object-Oriented Software Engineering*. s.l. : Prentice Hall, 1991, pp. 1-50.
3. **Meyer, Bertrand.** Applying "Design by Contract". *Computer (IEEE)*. October 1992, Vol. 25, 10, pp. 40-51.
4. *MDA Guide Version 1.0.1*. s.l. : OMG, June 2003.
5. **Cook, Steve, et al., et al.** *Domain-Specific Development with Visual Studio DSL Tools*. s.l. : Addison-Wesley Professional, 2007. 9780321398208.
6. **Frankel, David and Parodi, John.** *Using Model-Driven Architecture™ to Develop Web Services*. s.l. : IONA Technologies white paper, 2002.
7. **Warmer, Jos and Kleppe, Anneke.** *The Object Constraint Language: Getting Your Models Ready for MDA 2*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. 0321179366.
8. **Jézéquel, Jean-Marc, Train, Michel and Mingins, Christine.** *Design Patterns with Contracts*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000. 0201309599.
9. **Czarnecki, Krzysztof and Eisenecker, U.W.** *Generative Programming: methods, tools and applications*. London : Addison Wesley, 2000.
10. **Stahl, Thomas, Voelter, Markus and Czarnecki, Krzysztof.** *Model-Driven Software Development: Technology, Engineering, Management*. s.l. : John Wiley & Sons, 2006. 0470025700.
11. **Arnout, Karine.** *EiffelStudio: A Guided Tour*. 2001.
12. *Code Contracts for .NET: Runtime Verification and So Much More.* **Barnett, Mike.** s.l. : Springer, 2010.
13. *Embedded Contract Languages.* **Barnett, Mike, Fähndrich, Manuel and Logozzo, Francesco.** s.l. : Association for Computing Machinery, Inc., 2010.
14. *The Spec# programming system: An overview.* **Barnett, Mike, Leino, K. Rustan M. and Schulte, Wolfram.** s.l. : Springer, 2005.
15. **Microsoft.** *Code Contracts User Manual*. 2012.
16. **Liskov, Barbara and Wing, Jeanette.** A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. November, 1994, Vol. Vol. 16, 6.

17. *Static contract checking with Abstract Interpretation*. **Fähndrich, Manuel and Logozzo, Francesco**. s.l. : Springer Verlag, 2010. Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010).
18. **Flanagan, C., et al., et al.** Extended static checking for Java. *PLDI'02*. 2002.
19. **Barnett, M., et al., et al.** Boogie: A modular reusable verifier for object-oriented programs. *FMCO'05*. 2005.
20. **Iosif-Lazăr, Alexandru Florin and Sestoft, Peter**. *Assessment of Microsoft Code Contracts Usability for the C5 Generic Collection Library*. 2010.
21. **Pelechano, Vicente, et al., et al.** *Building tools for model driven development comparing Microsoft DSL tools and eclipse modeling plug-ins*. Barcelona : CIMNE, 2006.
22. **Venter, Barend H.** *Multi-language compilation*. 7,219,338 USA, May 15, 2007.
23. **Ng, Karen, et al., et al.** The Roslyn Project: Exposing the C# and VB compiler's code analysis. *MSDN*. [Online] October 2011. [Cited: 05 18, 2012.] <http://go.microsoft.com/fwlink/?LinkID=230702>.
24. *The igragh software package for complex network research*. **Csárdi, Gábor and Nepusz, Tamás**. s.l. : InterJournal Complex Systems, 2006.
25. *3D Visualization of Software Architectures*. **Feijs, Loe and de Jong, Roel**. s.l. : Communications of the ACM, 1998.
26. *EvoSpaces: 3D Visualization of Software Architecture*. **Alam, Sazzadul and Dugerdil, Philippe**. 2007. Int'l Conf. on Soft. Eng. and Knowledge Eng.
27. *The 4+1 View of Architecture*. **Kruchten, Philippe B.** s.l. : IEEE, 1995.
28. **Vanderseyppen, Francois M.** *Netron Graph Library Architecture*. 2004.
29. **Waldén, Kim and Nerson, Jean-Marc**. *Seamless Object-Oriented Software Architecture*. s.l. : Prentice Hall, 1994. 0130313033.
30. **Gries, David and Schneider, Fred B.** Avoiding the Undefined by Underspecification. *Computer Science Today*. 1995.
31. *Optimizing Development of a Complex Software by Using and Extending Design Patterns*. **Costantini, Fabien, Toinard, Christian and Chevassus, Nicolas**. s.l. : Citeseer, 2001.
32. *Improving UML designs using automatic design pattern detection*. **Bergenti, Federico and Poggi, Agostino**. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000). pp. 336-343.
33. *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. **Schmidt, Douglas C.** s.l. : Communications of the ACM, 1995.

34. **Özgür, Turhan.** *Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling (In the context of the Model-Driven Development)*. s.l. : School of Engineering Ronneby Sweden Blekinge Institute of Technology, 2007.

10. Appendices

Appendix 1 SolutionArchitectureLanguage NamespaceShape expanded.

[illegible]

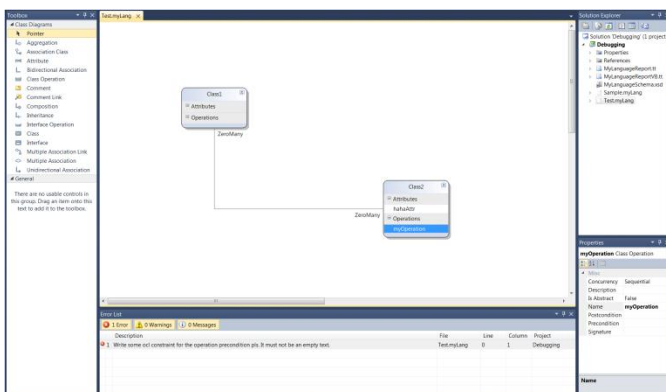
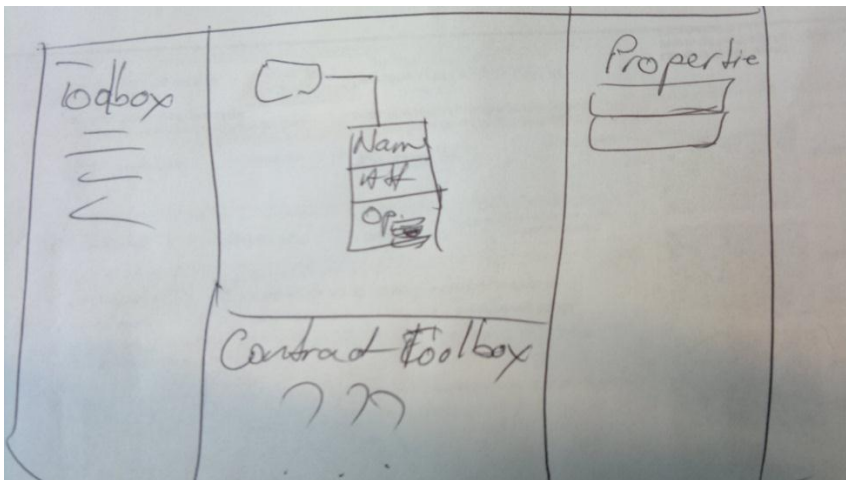
Appendix 2 FindBelowShapes

```

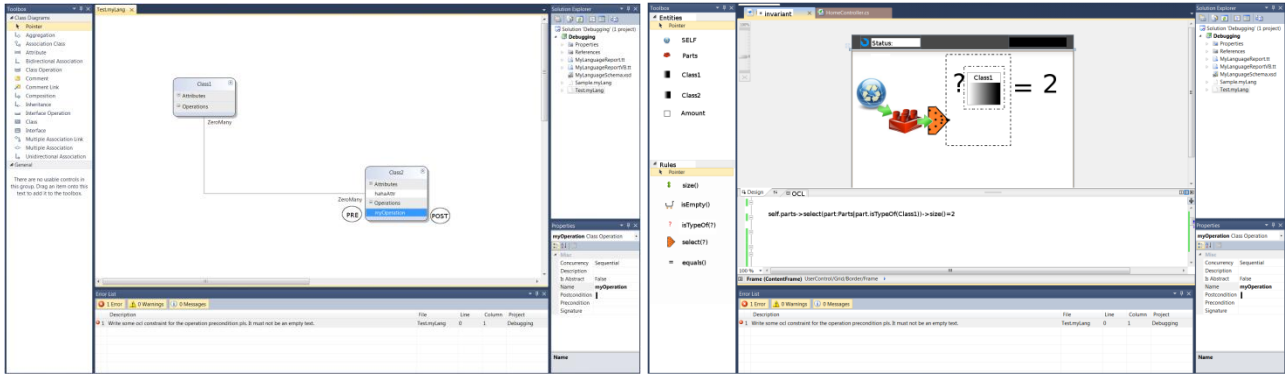
1. private List<ShapeElement> FindBelowShapes(NodeShape parentShape)
2. {
3.     var belowShapes = new List<ShapeElement>();
4.     foreach (var siblingShape in parentShape.NestedChildShapes)
5.     {
6.         if (siblingShape == this) continue;
7.         //determine if a shape is just a link - as links follow the connected shapes
8.         if ((siblingShape as LinkShape) != null) continue;
9.         if (siblingShape.BoundingBox.X > (Bounds.X + Bounds.Width)) continue;
10.        if ((siblingShape.BoundingBox.X + siblingShape.BoundingBox.Width) <
            Bounds.X)
11.            continue;
12.        if (siblingShape.BoundingBox.Y > Bounds.Y)
13.            belowShapes.Add(siblingShape);
14.    }
15.    return belowShapes;
16.}

```

Appendix 3 Prototypes



AMOC: A Visual Studio Extension for Application Modeling with Contracts



SolutionArchitectureLanguage1.sa

