



IciQL

The tutorial

Contents

1. Abstract	3
2. Goals.....	3
3. Base Technology.....	3
4. System Architecture	3
5. Editing experience and real-time results (WSIWYG).....	4
6. DSL Realtime Pipeline.....	4
7. The models	4
7.1 Models.....	5
7.1.1 QExp	5
7.1.2 QL	6
7.1.3 QLS.....	7
7.2 Grammars.....	8
7.2.1 QExp	8
7.2.2 QL	9
7.2.3 QLS.....	10
8. Generation templates	11
9. Model checking	12
10. Extra bonus: Projectional Editors for the same price.....	12
11. Acknowledges	13
12. References.....	13

Version	Date	Description	Author
1.0	2013.03.09	First draft version.	Pedro J. Molina, Icinetic Ruben J. Marrufo, Icinetic
1.1	2013.03.25	Added projectional editors and acks.	Ahmed Negm, Icinetic Ruben J. Marrufo, Icinetic Pedro J. Molina, Icinetic

1. Abstract

This document describes the Icinetic (<http://icineti.com>) implementation of the Language Workbenches Competition 2003 Challenge

http://www.languageworkbenches.net/index.php?title=LWC_2013 .

2. Goals

The main goals are:

- Implement the Expression, QL, and QLS.
- Provide a sample of how Icinetic used to address DSL construction
- Enjoy and improve our tools and process at the same time!

3. Base Technology

IciQL is implemented using the .NET 4.5 Framework and the C# language. All code is 100% managed code without any native extensions.

As far as .NET do not have a rich full modeling environment as Eclipse has (EMF, some modeling tools and infrastructure had to be developed ad-hoc.) Icinetic builds and maintain such set of tools for making MDD on the .NET platform robust and feasible.

The entry point of the modeling capabilities used to solve the challenge is call ONION.

ONION is a meta-modeling framework developed at Icinetic capable of:

- Define metamodels
- Create structures POJOs/POCOs for such model
- Provide serialization/deserialization facilities to different flavours
- Provide model validation, merge, diff and indexing facilities
- Provide model composition
- Lightweight and very fast designed for scalability.

4. System Architecture

IciQL comprises several modules:

- 1. WPF User Interface providing a rich text editor, error reporting and a embed web browser for rendering.
- 2. Model structures
- 3. Parsers
- 4. Model and type checking
- 5. Code generator / render

Component (1) UI is hand-coded but it is generic enough to be reused for several DSLs.

Models (2) and Parsers (3) are generated from meta-models. Extension points are provided to add custom queries.

Model checking (4) is partially provided by models, the rest is hand-coded: algorithmics do not pay off to be abstracted to a model, it can be implemented directly on C#.

Code generation (5) is purely declarative using StringTemplate 4.0 engine.

5. Editing experience and real-time results (WSIWYG)

Been able to directly see the result of the modeling effort is always helpful for the user.

In this way, IciQL was build thinking on providing this direct manipulation interface in real time. Fast parsing is a top level requirement to achieve this goal, and C# PEG parsing made it possible in IciQL without relying any low level libraries coded in C or C++ for example.

6. DSL Realtime Pipeline

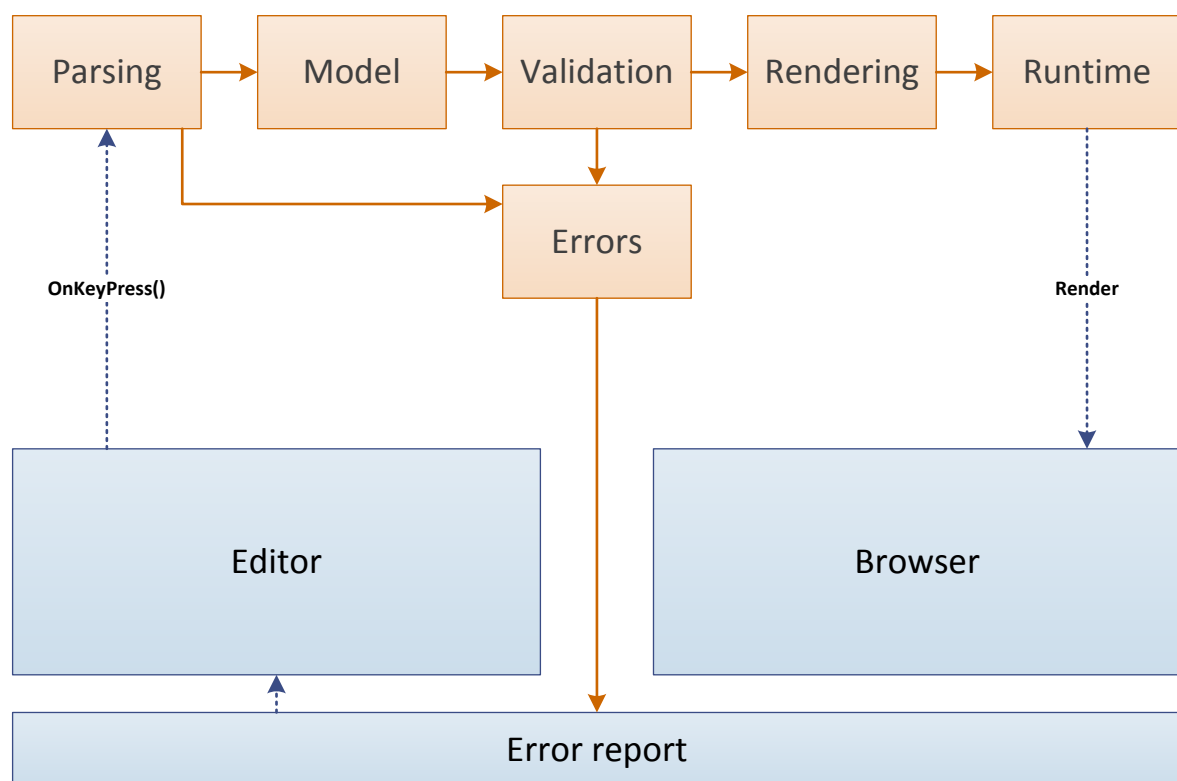


Figure 1. IciQL DSL realtime pipeline.

7. The models

The core of the DSLs are defined in models using Essential syntax. Then the ONION tooling converts them into working implementations via code generation.

As requested by the LWC2013, models were designed modular for reuse. Therefore three models were created:

- QExp for expressions
- QL for the Query Language
- QLS for the Style Language

7.1 Models

7.1.1 QExp

The specification for the **QExp** model follows. This model allows to represent tree expressions where the abstract Expression class is subclassed to add the different expressions supported.

QExp.meta

```
namespace IciQL.QExp
{
    abstract class Expression
    {}
    class BinaryExpression : Expression
    {
        string Operator;
        Expression Left;
        Expression Right;
    }
    class UnaryExpression : Expression
    {
        string Operator;
        Expression InnerExpression;
    }
    class ParenthesisExpression : Expression
    {
        Expression InnerExpression;
    }
    class TermExpression : Expression
    {
        string Identifier;
    }
    class LiteralExpression : Expression
    {
        string Literal;
    }
    class NumericExpression : Expression
    {
        string StringValue;
    }
    class TernaryExpression : Expression
    {
        Expression Condition;
        Expression PositiveExpression;
        Expression NegativeExpression;
    }
}

/*
BEGIN StructureGenerator DesignInfo

Root Expression
```

```

Query Expression : string PrintType
    return GetType().Name;
EndQuery

Query TermExpression : bool IsVariable
    return IciNetic.Onion.ValidationUtils.IsValidIdentifier(Identifier);
EndQuery

Query NumericExpression : decimal Value
    Decimal number;
    Decimal.TryParse(StringValue, out number);
    return number;
EndQuery

END StructureGenerator DesignInfo
*/

```

7.1.2 QL

QL contains the structure for form definitions. Note how Expressions are imported using the “using directive”.

QL.meta

```

using IciQL.QExp;

namespace IciQL.QL
{
    class Form
    {
        string Name;
        List<Block> Blocks;
    }
    class Block
    {}
    class Question : Block
    {
        string Name;
        string Label;
        //QuestionType Type; //Calculated
        string Type;
        Expression? Calculation;
    }
    enum QuestionType
    {
        None,
        Boolean,
        String,
        Integer,
        Date,
        Decimal,
        Money
    }
    class ClauseBlock : Block
    {
        Expression Condition;
        List<Block> Blocks;
    }
}

```

```

        /* Expressions: Imported from: IciQL.QExp; */
    }

    /*
    BEGIN StructureGenerator DesignInfo
    Root Form

    LocalId Form Name
    LocalId Question Name

    GlobalId Form Name
    GlobalId Question Name

    Merge Form MergeLeft

    Query Question : QuestionType QuestionType
        QuestionType qt;
        if (Enum.TryParse(Type, true, out qt))
            return qt; //Value is OK
        return QuestionType.None; //error (undef)
    EndQuery

    Query Question : bool IsValidType
        return QuestionType != QuestionType.None;
    EndQuery

    END StructureGenerator DesignInfo
    */

```

7.1.3 QLS

Finally QLS adds the concepts needed for Style definition reusing concepts from QExp and QL when needed.

QLS.meta

```

using IciQL.QExp;
using IciQL.QL;

namespace IciQL.QLS
{
    class FormStyle
    {
        Form Form;
        Style? Style;
    }

    class Style
    {
        string Name;
        List<StyleDef> StyleDefinitions;
    }
    abstract class StyleDef
    {
        string Name;
    }
    class QuestionStyle : StyleDef
    {

```

```

        List<StyleProperty> StyleProperties;
    }
    class SectionStyle : StyleDef
    {
        string? Label;
        List<StyleDef> StyleDefinitions;
    }
    class StyleProperty
    {
        string Key;
        Expression Value;
    }
}

/*
BEGIN StructureGenerator DesignInfo

Root FormStyle

LocalId Style Name
LocalId StyleDef Name
LocalId StyleProperty Key

GlobalId Style Name
GlobalId StyleDef Style[GlobalId].StyleDefinitions[Name]

Merge Style MergeLeft

END StructureGenerator DesignInfo
*/

```

7.2 Grammars

DSL textual grammar for the languages are also specified in three reusable and composable grammar specifications.

7.2.1 QExp

The parsing for expressions is as basic as needed for LWC. The grammar can/will be enhanced to support more operators and priority. The ONION grammar DLS allows to create production rules with a friendly syntax (a la BNF and PEG) and add declarative sentences (\Rightarrow to build the three, the underlying model).

QExp.gram

```

language IciQL.QExp

modelNameSpace IciQL.QExp
whitespace interleaved
singleLineComment //
multiLineComment /* */

ExpressionMain      ::=      Expression ;

Expression           ::=      TernaryExpression / Expression2 ;

```



```

TernaryExpression ::= co:Expression2 TERNARYOP t:Expression COLON f:Expression
                    => TernaryExpression { Condition = co,
                    PositiveExpression=t, NegativeExpression=f };

Expression2 ::= UnaryExpression / BinaryExpression / BasicExpression ;

UnaryExpression ::= op:UnaryOperator inner:Expression
                  => UnaryExpression { Operator=op, InnerExpression=inner } ;

BinaryExpression ::= l:BasicExpression op:Operator r:Expression
                   => BinaryExpression { Left=l, Operator=op, Right=r } ;

BasicExpression ::= ParenthesisExpression / LiteralExpression / Term /
                    NumberExpression ;

ParenthesisExpression ::= LPAREN exp:Expression @RPAREN
                        => ParenthesisExpression { InnerExpression = exp } ;

NumberExpression ::= ne:Number
                  => NumericExpression { StringValue = ne } ;

LiteralExpression ::= QUOTE str:STRING_CONTENT @QUOTE
                   => LiteralExpression { Literal = str } ;

Term ::= id:ID => TermExpression { Identifier = id } ;

token UnaryOperator ::= MINUS / NOT ;

token Operator ::= PLUS / MINUS / DIVIDE / MULTIPLY / GT / LT / GTE / LTE / EQ /
NEQ ;

token Number ::= [0-9]+ ;

token ID ::= [A-Za-z_][A-Za-z_0-9]* ;

token GT ::= '>';
token LT ::= '<';
token GTE ::= '>=';
token LTE ::= '<=';
token EQ ::= '==';
token NEQ ::= '!=';

token NOT ::= '!';

token PLUS ::= '+';
token MINUS ::= '-';
token DIVIDE ::= '/';
token MULTIPLY ::= '*';
token LPAREN ::= '(';
token RPAREN ::= ')';
token QUOTE ::= '"';
token TERNARYOP ::= '?';
token COLON ::= ':';
token SEMI ::= ';';

token STRING_CONTENT ::= [#x20-#x21#x23-#xFFFF]* ;
token STRING_LITERAL ::= ~QUOTE [#x20-#x21#x23-#xFFFF]* ~QUOTE ;

```

7.2.2 QL

Similarly, QL model provides the syntax needed for form specification:

QL.gram

```
language IciQL.QL

modelNameSpace IciQL.QL

whitespace interleaved
singleLineComment //
multiLineComment /* */

includeLanguage IciQL.QExp

Form          ::=      FORM name:@ID @LBRACKET bi:BlockItem* @RBRACKET
                  => Form { Name = name, Blocks += bi } ;

BlockItem     ::=      IfClause / Question ;

Question ::=      na:ID COLON @QUOTE str:STRING_CONTENT @QUOTE ty:@ID (LPARAM calc:Expression
@RPARAM)?
                  => Question { Name = na, Label = str, Type= ty, Calculation = calc} ;

IfClause ::=      IF @LPARAM exp:Expression @RPARAM @LBRACKET bi:BlockItem* @RBRACKET
                  => ClauseBlock { Condition = exp, Blocks += bi } ;

token FORM          ::=      'form';
token LPARAM        ::=      '(';
token RPARAM        ::=      ')';
token LBRACKET      ::=      '{';
token RBRACKET      ::=      '}';
token IF            ::=      'if';
```

7.2.3 QLS

Finally, grammar for style follows:

QLS.gram

```
language IciQL.QL

modelNameSpace IciQL.QL

whitespace interleaved
singleLineComment //
multiLineComment /* */

includeLanguage IciQL.QExp

Form          ::=      FORM name:@ID @LBRACKET bi:BlockItem* @RBRACKET
                  => Form { Name = name, Blocks += bi } ;

BlockItem     ::=      IfClause / Question ;

Question ::=      na:ID COLON @QUOTE str:STRING_CONTENT @QUOTE ty:@ID (LPARAM calc:Expression
@RPARAM)?
                  => Question { Name = na, Label = str, Type= ty, Calculation = calc} ;

IfClause      ::=      IF @LPARAM exp:Expression @RPARAM @LBRACKET bi:BlockItem* @RBRACKET
```

```

=> ClauseBlock { Condition = exp, Blocks += bi } ;

token FORM      ::=      'form';
token LPARAM    ::=      '(';
token RPARAM    ::=      ')';
token LBRACKET  ::=      '{';
token RBRACKET  ::=      '}';
token IF        ::=      'if';

```

8. Generation templates

StringTemplate is clean and fast template engine for code generation. The HTML, CSS and Javascript rendering is fully implemented using StringTemplate: Some extract of the templates follows:

html5.stg

```

group html5;
delimiters "$", "$"

genForm(model, form, style)::=<<
$genHtmlHeader(form)$
$form.Blocks:genBlock()$
$genHtmlTail(form)$
>>

genHtmlHeader(form)::=<<
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>$form.Name$</title>
  <style>
    body { font-family: Calibri; }
    tt { font-family: Consolas, Monospaced, 'Courier New'; color: #555; }
    .question {
      background-color: ff3;
      border: 1px solid #fc0;
    }
    //...
  </style>
</head>
<body onLoad="init()">
  <h1>Form $form.Name$</h1>
>>

genHtmlTail(form)::=<<
</body>
</html>
>>

genBlock(it)::=<<
$if(it.IsQuestion)$genQuestion(it)$endif$
$if(it.IsClauseBlock)$genClauseBlock(it)$endif$
>>

genQuestion(it)::=<<
<div id="question_${it.Name$}" class="question">

```

```

<div class="name">${it.Name$}</div>
<div class="label">${it.Label$}</div>
<div class="entryfield">
  $if(it.IsBoolean)$<input type="checkbox" id="${it.Name$}" class="boolean" value="true"
onClick="window.setTimeout( function(){ valueChanged('${it.Name$}') },1)"/>$endif$
  $if(it.IsInteger)$<input type="text" id="${it.Name$}" class="integer" value=""
onKeyUp="window.setTimeout( function(){ fieldChanged('${it.Name$}', 'integer') },1)"/>$endif$
  $if(it.IsMoney)$<input type="text" id="${it.Name$}" class="money" value=""
onKeyUp="window.setTimeout( function(){ fieldChanged('${it.Name$}', 'money') },1)"/>$endif$
  $if(it.IsString)$<input type="text" id="${it.Name$}" class="string" value=""
onKeyUp="window.setTimeout( function(){ fieldChanged('${it.Name$}', 'string') },1)"/>$endif$
  $if(it.IsDate)$<input type="text" id="${it.Name$}" class="date" value=""
onKeyUp="window.setTimeout( function(){ fieldChanged('${it.Name$}', 'date') },1)"/>$endif$
  $if(it.IsDecimal)$<input type="text" id="${it.Name$}" class="decimal" value=""
onKeyUp="window.setTimeout( function(){ fieldChanged('${it.Name$}', 'decimal') },1)"/>$endif$
  <span class="inlineValidation hidden" id="validationMessage_${it.Name$}"></span>
</div>
</div>
>>

genClauseBlock(it)::=<<
<h2>If Clause</h2>
<p>
  <tt>${it.Condition$}</tt>
  <div class="block">
    ${it.Blocks:genBlock()}$
  </div>
</p>
>>

...

```

9. Model checking

Essential metamodels provides built-in support for optional/mandatory, cardinality and unique id checking. Extra validations can be added as custom queries and checking via model queries, or via custom code.

TypeChecking is implemented using a classical tree visit (bottom-up) (from leaves to roots) calculating the computed type and reporting errors when expressions are not compatible.

Cycle dependence checking was delegated to a precalculation in a graph class with nodes. This class contains a custom algorithm to check and found cycles in the graph. If cycles in the graph were found, a dependency cycle was reported.

10. Extra bonus: Projectional Editors for the same price

Lastly, thanks to the great work of Ahmed Negm in the latest weeks, we were able to add in this solution a second type of editors: the projectional ones.

WPF is a great presentation layer for creating projectional editors, and this challenges allows us to test them in deep. Projectional editors avoids the usage of parsing, therefore we were able to directly data-bind with our models and reuse the rest of the pipeline.

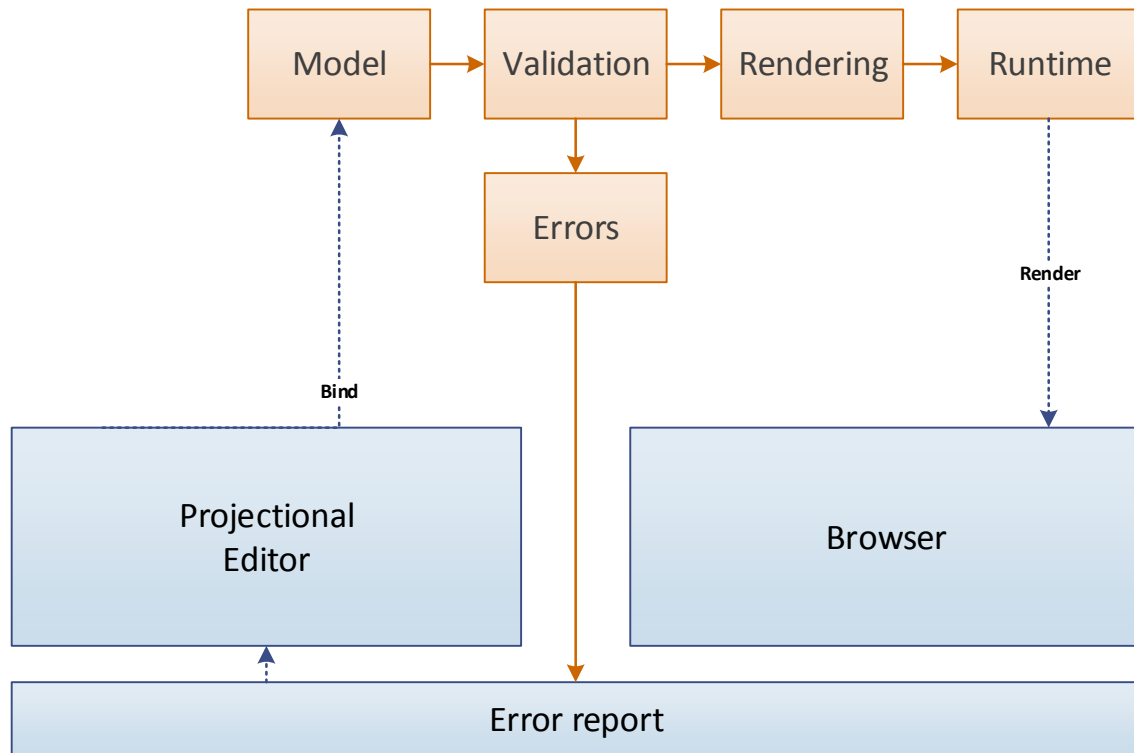


Figure 2. Architecture and processing pipeline with Projectional Editors.

11. Acknowledges

We: Ruben, Ahmed and Pedro J. wants to thank you to all the Icinetic team that spend many time to beta-test and improve with ideas this implementation and also to the LWC2013 organizers and other teams for encouraging us to push forwards our owns limits.

12. References

- LWC2013 http://www.languageworkbenches.net/index.php?title=LWC_2013
- Code Generation 2013 <http://www.codegeneration.net/cg2013/index.php>
- Essential: <http://pjmolina.com/metalevel/essential/>
- Icinetic: <http://icineti.com>