Language Workbenches Competition 2011

Essential submission

Pedro J. Molina, PhD
Valencia, Spain
pjmolina(at)gmail.com
http://pjmolina.com/metalevel
@pmolinam

Version	Date	Description
1.00	25th February 2011	First version

Abstract

This document represents the submission to the Language Workbenches Competition 2011 (www.languageworkbenches.net) using the Essential tool. The main objective is to show how the concepts and the intended usage of the tool with a reference sample to be compared with other Language Workbenches.

Content

Introduction

The **Language Workbench Competition** is an initiative to show the capabilities of different tools available to materialize the concept of a **Language Workbench** as proposed by Martin Fowler (www.languageworkbenches.net).

This document is the submission to LWC11 created to illustrate how the **Essential** tool can be used to accomplish the challenges proposed in the competition.

Essential (pjmolina.com/metalevel/essential) is a tool created by Pedro J. Molina to explore the creation of models and meta-models with an agile approach based on meta-model interpretation.

The main motivation behind the construction this tool comes from the frequent lack of agility found in traditional MDD tools that forces developers to a long generate-build-debug cycles in order to develop tailored MDD tools for concrete domains. Essential promotes building and prototyping the tools as fast as possible.

The rest of the document follows exactly the organization proposed in www.languageworkbenches.net/LWCTask-1.0.pdf to make easier the comparison with other tools.

Latest version to this document and code sample can be obtained at:

http://code.google.com/p/lwc11-essential/

Phase 0. Basics

0.1 Simple (structural) DSL without any fancy expression language or such

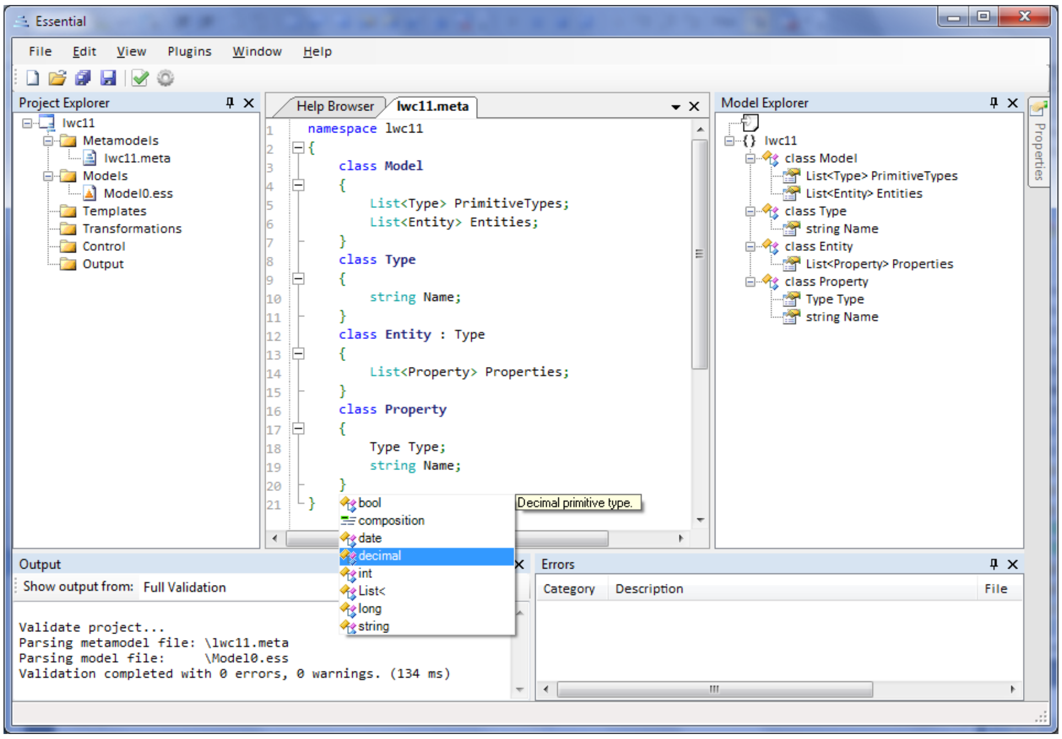
Challenge: *Build a simple data definition language to define entities with properties. Properties have a name and a type. It should be possible to use primitive types for properties, as well as other Entities.*

```
entity Person {
  string name
  string firstname
  date birthdate
  Car ownedCar
}

entity Car {
  string make
  string model
}
```

Proposed solution:

1. Open Essential and create a new project called LWC11.



```
namespace lwc11
{
    class Model
    {
        List<Type> PrimitiveTypes;
        List<Entity> Entities;
    }
    class Type
    {
        string Name;
    }
    class Entity : Type
    {
        List<Property> Properties;
    }
    class Property
    {
        Type Type;
        string Name;
    }
}
```

```
Validate project...
Parsing metamodel file: \lwc11.meta
Parsing model file:     \Model0.ess
Validation completed with 0 errors, 0 warnings. (134 ms)
```

Figure 1. Essential IDE after the definition of the metamodel.

2. Create a new meta-model file called **lwc11.meta** and add the following definitions (see code-completion (via `[Control] - [Space]`, syntax coloring, a model explorer tree, and in-place error reporting as you type):

```
namespace lwc11
{
    class Model
    {
        List<Type> PrimitiveTypes;
        List<Entity> Entities;
    }
    class Type
    {
        string Name;
    }
    class Entity : Type
    {
        List<Property> Properties;
    }
    class Property
    {
        Type Type;
        string Name;
    }
}
```

3. Now, create a new model file called **Model0.ess** and create the sample model. Again, note about code-completion, syntax coloring, model explorer and in-place error reporting as you type:

```
using lwc11;
namespace lwc11.model0
{
    Model m0
    {
        PrimitiveTypes = [string, date];
        Entities = [Person, Car];
    }
    //Define primitive types as needed
    Type string     {
        Name = "string";
    }
    Type date     {
        Name = "date";
    }
    Entity Person
    {
             Name = "Person";
             Properties = [
                        Property pe1
                        {
                                   Name = "name";
                                   Type = string;
                        },
                        Property pe2
                        {
                                   Name = "firstname";
                                   Type = string;
                        },
                        Property pe3
                        {
                                   Name = "birthdate";
                                   Type = date;
                        },
                        Property pe4
                        {
                                   Name = "ownedCar";
                                   Type = Car;
                        }
             ];
    }
    Entity Car
    {
             Name = "Car";
             Properties = [
                        Property p0
                        {
                                   Name = "make";
                                   Type = string;
                        },
                        Property p1
                        {
                                   Name = "model";
                                   Type = string;
                        }
             ];
    }
}
```

4. Validate the model and check against the meta-model its validity pressing the button marked as "Validate All".

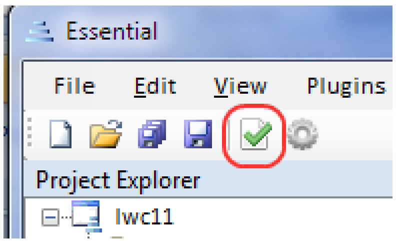


Figure 2. Location of the "Validate All" button.

0.2 Code generation to GPL such as Java, C#, C++ or XML

Code generation in Essential is accomplished using templates and control files to transform the model into code. In this case, Java POJOs will be generated as a quick proof of concept.

1. To create a code generator for the model, add a new workflow/control file called **Control.ctl** and add the following code:

```
using lwc11;
using lwc11.model0;

namespace lwc.javaGenerator
{
    transformation Main()
    {
        Model model = lwc11.model0.m0;
        foreach(Entity ent in model.Entities)
        {
            javaPojo.genPojo(ent) > "$ent.Name; format="Pascalcase"$.java";
        }
    }
}
```

   This control file implements a DSL for driving the code generation:

   - selecting an element of the model,
   - iterating,
   - and finally, applying a template per Entity and selecting the output filename and location.

   Note about the integrated expression for the generation of the filename.

2. Add a new template file called **javaPojo.stg** and add the following code:

```
group javaPojo;

genPojo(entity) ::= <<
import java.lang.*;
import java.io.*;
import java.util.*;

public class $entity.Name$ implements Serializable {
          //Private members
          $entity.Properties:genPrivateMember(); separator="\n"$

          //Public accessors
          $entity.Properties:genMemberAccessors(); separator="\n\n"$
}
>>

genPrivateMember()::= <<
private $it.Type:genType()$ $it.Name; format="camelcase"$;
>>

genType()::= <<$typeMapper.(it.Name)$>>

genMemberAccessors()::= <<
public $it.Type:genType()$ get$it.Name; format="pascalcase"$() {
        return $it.Name; format="camelcase"$;
}
public void set$it.Name; format="pascalcase"$($it.Type:genType()$ value) {
        this.$it.Name; format="camelcase"$ = value;
}
>>

typeMapper ::=
[
    "string" : "String",
    "date"   : "Date",
    default  : key
]
```

Essential used al well known template language created by Terrence Parr and called *StringTemplate* (www.stringtemplate.org). StringTemplate syntax and capabilities is well documented in the StringTemplate website. Essential provides colored syntax editor for in-place StringTemplate edition.

3. Save all and, in the Project Explorer, select the control file. Open the contextual menu with the secondary button of the mouse and select **Apply Transformation**.
   The code generation will produce the output files for Java code showing the following log report.

```
Output base directory: C:\lwc2011-essential\Task 0.2\bin
Executing transformation Main()
 Model model = lwc11.model0.m0;
 foreach (Entity ent in model.Entities)
  Apply javaPojo.genPojo() > [Person.java]
  Apply javaPojo.genPojo() > [Car.java]
Transformed in 1325 ms.
```

4. To review the contents of the produced files, select the folder node **Output** on the Project Explorer and open the contextual menu with the secondary button of the mouse and select **Explore Output Directory**. A new file explorer window will show the directory and generated files containing the following contents for **Person.Java**:

```
import java.lang.*;
import java.io.*;
import java.util.*;

public class Person implements Serializable {
  //Private members
  private string name;
  private string firstname;
  private date birthdate;
  private Car ownedCar;

  //Public accessors
  public string getName() {
        return name;
  }
  public void setName(string value) {
        this.name = value;
  }

  public string getFirstname() {
        return firstname;
  }
  public void setFirstname(string value) {
        this.firstname = value;
  }

  public date getBirthdate() {
        return birthdate;
  }
  public void setBirthdate(date value) {
        this.birthdate = value;
  }

  public Car getOwnedCar() {
        return ownedCar;
  }
  public void setOwnedCar(Car value) {
        this.ownedCar = value;
```

```
    }
}
```

And **Car.java**:

```
import java.lang.*;
import java.io.*;
import java.util.*;

public class Car implements Serializable {
          //Private members
          private String make;
          private String model;

          //Public accessors
          public String getMake() {
                 return make;
          }
          public void setMake(String value) {
                 this.make = value;
          }

          public String getModel() {
                 return model;
          }
          public void setModel(String value) {
                 this.model = value;
          }
}
```

0.3 Simple constraint checks such as name-uniqueness

Unique names for definitions (references) are provided out-of-the-box based on the namespace and concept declaration provided by Essential by itself.
Each namespace defines a naming context similar to UML or Java packages or XML or .NET namespaces.

Custom constraints **are not supported in the current version** of the tool, but it could be a nice enhancement for next releases if needed using a declarative syntax like this one:

```
namespace lwc11
{
    class Entity
    {
       string Key;
    }

    check unique Entity.Key;
}
```

Or a more powerful and expressiveness like this other one based on instance selection and expression evaluation:

```
namespace lwc11
{
    class Entity
    {
       string Key;

       check uniqueEntityKey
       {
            (Collect(Entity.Instances) where (item.Key == this.Key)).Count == 1;
       }
    }
}
```

0.4 Show how to break down a (large) model into several parts, while still cross-referencing between the parts

Essential was created targeting scalability of the models. Models can be broken down using **partial definitions**. Therefore, you can split the previous models in various files and Essential will load all of them and check the consistency of the composite model.

Example A: create a new metamodel called **metaExtension1.meta** and add the following definitions:

```
namespace lwc11
{
    class Entity
    {
       string Label;
    }
    class Property
    {
       int? Size;
    }
}
```

Save the file. This extends the definition of the metamodel adding a compulsory named **Label** property to **Entity** and an optional **Size** attribute to the **Property**.

As soon as the file is saved, the metamodel has been extended with the new definition. Now, the model is invalid due to no compulsory Label is provided for each entity.

Fix the models adding a property Label for each Entity. Optionally, add a Size property if needed to any Property. Use `[Control]-[Space]` to check with auto-completion if the metamodel was property extended.

Example B: Similar to metamodels, models can also be partitioned in different files. For example a sensible partitioning for a growing model could be storing primitives types in a file called: **model0.ess**

```
using lwc11;

namespace lwc11.model0
{
  //Define primitive types as needed
    Type string
    {
        Name = "string";
    }
    Type date
    {
        Name = "date";
    }
}
```

Grouping entitity definition in a second file **model1.ess**

```
using lwc11;

namespace lwc11.model0
```

```
{
            //Entities
    Entity Person
    {
        ...
    }
    Entity Car
    {
        ...
    }
}
```

And finally a model root in **model2.ess**:

```
using lwc11;

namespace lwc11.model0
{
    Model m0
    {
        PrimitiveTypes = [string, date];
        Entities = [Person, Car];
    }
}
```

Transformations namespaces are also be splitted and merged in the same way. For templates, StringTemplates provides group inheritance and interfaces allowing to reuse all the templates defined in parent group context.

Phase 1 - Advanced

1.1 Show the integration of several languages

Essential allows reusing and combining meta-models and model definitions with the usage of the "**using**" clause and **partial definitions**. After importing a meta-model or model definition, the definitions are available to be used inside the language.

This kind of integration can be seen as a form of metamodeling extension introduced in previous task 0.4.

Example: Introducing a model for containing settings for code generation to select Java or C# code generation or any other output platform setting.

1. Create a new minimal metamodel for the settings called **settings.meta**:

```
namespace settings
{
    enum Language
    {
      Java,
      CSharp
    }
    class GenerationSettings
    {
                Language TargetLanguage;
    }
}
```

2. Create a new model for the settings:

```
using settings;
namespace generationSettings
{
  GenerationSettings sets
  {
      TargetLanguage = Language.Java;
  }
}
```

3. Finally, we can use the new model in the controller **control.ctl** to drive the code generation if the generation switch is on for Java:

```
using lwc11;
using lwc11.model0;
using settings;

namespace lwc.javaGenerator
{
    transformation Main()
    {
        Model model = lwc11.model0.m0;
        GenerationSettings set = generationSettings.sets;

        foreach(Entity ent in model.Entities)
        {
             if (set.TargetLanguage == Language.Java)  //Conditional generation based in a Settings model
             {
              javaPojo.genPojo(ent) > "$ent.Name; format="camelcase"$.java";
             }
          }
    }
}
```

Note: Currently, the concrete syntax used for specifying the models is prefixed and derived automatically by the metamodel. Others tools allows creating editors or grammars to provide degrees of freedom here. For the moment in Essential, the main focus is to have a flexible environment and easy way of transforming such models. A secondary objective will be to introduce improved model edition capabilities.

1.2 Demonstrate how to implement runtime type systems

As exemplified before during the previous examples, Essential provides the following primitive types:

Primitive type	Semantics
`string`	Unrestricted text.
`bool`	Logic value: `true` or `false`.
`int`	Integer.
`long`	Long integer.
`decimal`	Decimal value. Useful for money and when discrete numeric precission is a must.
`char`	Character.
`date`	Constains a date in the W3C format (locale independent): #yyyy-MM-dd#
`time`	Contains a time expression given the W3C format: #hh:mm:ss(((+|-)hh:mm)|Z)?#
`datetime`	Constains a datetime value in the W3C format (locale independent): #yyyy-MM-ddThh:mm:ss(TZ)?#

And the following Abastract Data Types (ADT) constructors:

TAD	Semantics
`List<T>`	List of type T
`Dictionary<K, T>`	Dictionary of type K as key and type T as values.

The habitual cardinality operators when defining properties apply:

Operator	Cardinality Semantics
`(nothing)`	`(1..1)` Compulsory univaluated for simple types and `(0..*)` for Lists.
`?`	`(0..1)` Optional. Element can be present or ausent.
`+`	`(1..*)` Compulsory and multivaluated.
`*`	`(0..*)` Optional and multivaluated.
`0..*`	`(0..*)` Same as previous.
`n..m`	`(n..m)` Multivaluated with a minimun of (n) and a maximus of (m).

- All of them can be used in metamodeling construction.
- TADs can nested (e.g. List<List<Person>>).
- Inheritance is also implemented and checked at the class level.

1.3 Show how to do a model-to-model transformation

Essential provides a specific DSL for describing Model-to-Model transformation. The following sample will transform the sample Entity model to a Relational Table based model.

1. Add a new metamodel for a basic data base schema model called **db.meta**:

```
namespace db
{
    class Schema
    {
            List<Table> Tables;
    }
    class Table
    {
            string Name;
            List<Column> Columns;
    }
    class Column
    {
            string Name;
            string Type;
    }
}
```

2. Add a new transformation (M2M) called **DeriveDatabase.mtrx**:

```
using lwc11;
using db;

namespace deriveDatabase
{
    m2m_transformation Schema DeriveDB(Model m)
    {
            Rule1: Schema s << Model m
            {
                    s.Name = "ModelPersistence";
                    s.Tables = [ Table << forall m.Entities ];
            }

            Rule2: Table t << Entity e
            {
                    t.Name = e.Name;
                    t.Columns = [ Column << forall e.Properties ];
            }

            Rule3: Column c << Property p
            {
                    c.Name = p.Name;
                    c.Type = p.Type.Name;
            }
    }
}
```

   This DSL implements a rule language to declarative transform an input model into in a second one. In this case, we are translating an entity to a table, and a property to a column.

3. To test it, change the control file **control.ctl** to reflect the following code:

```
using lwc11;
using lwc11.model0;
using db;
using deriveDatabase;

namespace control
{
     transformation Main()
     {
               Model model = m0;

               m0 > "sourceModel.ess";
               Schema s0 = DeriveDB(m0);
               s0 > "schemaModel.ess"
     }
}
```

The “>” operator is used to serialize a model to file. This is quite useful to debug and see what going on with the model.
The call `Schema s0 = DeriveDB(m0);` invokes the M2M transformation, and creates a new model. The next line dumps it to file.

4. Finally, apply the transformation (right click into **control.ctl** and press **Apply Transformation**.
5. The output file (**schemaModel.ess**) will contains:

```
using db;
namespace Default
{
          Column R0
          {
                    Name = "name";
                    Type = "string";
          }
          Column R1
          {
                    Name = "firstname";
                    Type = "string";
          }
          ...
          Table R4
          {
                    Name = "Person";
                    Columns = [R0, R1, R2, R3];
          }
          Column R5
          ...
          Table R7
          {
                    Name = "Car";
                    Columns = [R5, R6];
          }
          Schema R8
          {
                    Name = "ModelPersistence";
                    Tables = [R4, R7];
          }
}
```

Using this transformed model to generate DLL SQL is an easy Model-2-Text task.

1.4 Some kind of visibility/namespaces/scoping for references

As commented before, Essential provides out-of-the-box a scoping mechanism based on namespaces. All object references are created inside a namespace scope. Referencing can be resolved using full qualified names or importing namespaces (in the same way as you could import packages in Java or namespaces in .NET or XML).

No extension mechanism is implemented currently for adding additional or custom forms of scoping.

1.5 Integrating manually written code (again in Java, C# or C++)

Declarative models and transformation are preferred when possible because avoids technology locking and prevents portability. However in some occasions algorithms implemented in code can help in the transformation mode.

Essential is implementing **user escapes** as an experimental feature to allow users to create a folder called `\userEscapes\` behind the model root folder. Being Essential a .NET application, calling dynamic .NET code and on the fly compilation of C# or VB.NET is a simple task.

The idea behind the scenes is to provide C# source or binary assemblies code implementing a library and placed in this special folder.

On rutime, Essential compiles and load all the libraries found and expose such libraries as functions to the modeling environment.

A sample library follows in C# implementing sample operations:

```
using System;

namespace LibDemo
{
    public class LibDemo
    {
        public int Add(int a, int b)
        {
            return a + b;
        }
        public string Echo(string msg)
        {
            return msg;
        }
    }
}
```

After this library import, the operation `LibDemo.Add(a,b)` and the operation `LibDemo.Echo(msg)` will be available* in the modeling environment.

***NOTE:** At the time of writing, this is an experimental feature: currently is not available in all modeling contexts.

1.6 Multiple generators

Another generator using the same model could address a second platform using a different language. For example we will generate C# POCOs (Plain Old CRL objects, aka the .NET equivalent for POJOs) but you can do the same for any other language of your choice.

1. The first step is to create a new template group targeting .NET code. To do that create a new template file called **csharpPoco.stg** and add the following definitions:

```
//csharpPoco.stg
group csharpPoco;

genPoco(entity) ::= <<
using System;
using System.Collection.Generics;

namespace Demo.Pocos
{
          [Serializable]
          public class $entity.Name; format="pascalcase"$
          {
            public $entity.Name; format="pascalcase"$() {}

            //Private members
            $entity.Properties:genPrivateMember(); separator="\n"$

            //Public accessors
            $entity.Properties:genMemberAccessors(); separator="\n\n"$
          }
}
>>

genPrivateMember()::= <<
private $genType(it.Type)$ $it.Name; format="camelcase"$;
>>

genType()::= <<$typeMapper.(it.Name)$>>

genMemberAccessors()::= <<
public $genType(it.Type)$ $it.Name; format="pascalcase"$
{
          get { return $it.Name; format="camelcase"$; }
          set { this.$it.Name; format="camelcase"$ = value; }
}
>>

typeMapper ::=
[
            "string" : "string",
            "Date"   : "DateTime",
            default  : key
]
```

The file adds the needed templates to generate C# classes and members. The **typeMapper** is hash-like facility to translate abstract types (model types) (left side of the map) to its correspondent C# type implementations (right side).

2. The second and last step is to use these templates from the control of code generation in a Model-to-Text transformation. Open the **Control.ctl** file and add the control code marked in bold:

```
using lwc11.meta;
using lwc11.model0;

namespace lwc.javaGenerator
{
    transformation Main()
    {
        Model model = lwc11.model0.m0;

        foreach(Entity ent in model.Entities)
        {
            //Java code generation
            javaPojo.genPojo(ent) > "src/java/$ent.Name; format="pascalcase"$.java";

            //C# code generation
            csharpPoco.genPoco(ent) > "src/cs/$ent.Name; format="pascalcase"$.cs";
        }
    }
}
```

Phase 2 - Non-Functional

2.1 How to evolve the DSL without breaking existing models

Evolving the DSL can be done using diverse techniques. Depending on the size of the models, the number of models and the effort that every choice has, we will probably have from one alternative or another:

1. Adding an optional definition will not break old models (similar to versioning Web Services with WSDL). This choice has for sure shortcomings but is the only option if models need to be backwards compatible.

2. A more general approach is **versioning the meta-model**: M1 and M2 and **create a M2M (model to model) transformation** to migrate models from version 1 to version 2. The task for accomplish this were discussed described when describing Model-to-Model transformations.

2.2 How to work with the models efficiently in the team

All files in Essential projects are based on text formats and UTF-8 text files. Therefore, integration with version control systems is direct and easy. Moreover, it is strongly recommended to do it that way, as the model and code generation is now our main development asset. Integration with Mercurial, Git, or Subversion is natural choice for store and share version of the models.

The split and merge capabilities of Essential allows partitioning the model in several files depending of your needs. Partitioning your models and applying Separation of Concerns is recommended to split big models in more manageable smaller files.

2.3 Demonstrate Scalability of the tools

The tool has been tested for a year and half to model and generate domain models with more than 120 classes and able to generate 140 tables, NHibernate mappings, POCO/POJO, DTOs, service layer and a web service layer. All generated in less than 45 seconds.

A command line version of the tool exists to allow integration with automatic builds and continuous integration (CI) systems.

In the reverse engineering field, diverse legacy Oracle Developer application (with an average about 140 forms, with an average of 20 controls per form and 15 events per form) code has been explored and retrieved information for pattern matching, model extraction and latter code generation to Java.

Phase 3 - Freestyle

3.1 Reflection

Essentials provides in-built level of **Reflection**. This allows to describe a metamodel in terms of a model itself. This opens the door to new possibilities and extensions. Let's see an example. The base sample metamodel is the following one:

```
namespace lwc11
{
    class Model
    {
        List<Type> PrimitiveTypes;
        List<Entity> Entities;
    }
    class Type
    {
        string Name;
    }
    class Entity : Type
    {
        List<Property> Properties;
    }
    class Property
    {
        Type Type;
        string Name;
    }
}
```

But the core primitives in Essential (classes, properties, enums, etc.) can be represented as a metamodel in the following form:

```
namespace Essential.Metamodel
{
    class EssentialModel
    {
           string Name;
           string GenerationTimestamp;
           List<Namespace> Namespaces;
    }
    class Namespace
    {
           string Name;
           List<Class> Classes;
           List<Enum> Enums;
    }
    class Class
    {
           string Name;
           List<Property> Properties;
    }
    class Property
    {
           string Name;
           string Type;

                      ...
    }
    class Enum
    {
               string Name;
               List<EnumItem> Items;
    }
    class EnumItem
    {
           string Key;
           int Order;
    }
}
```

Our sample metamodel can be seen in a reflective way like this one:

```
using Essential.Metamodel;
namespace Model
{
  EssentialModel model
  {
            Name = "name";
            GenerationTimestamp = "2011-02-25T00:08:01";
            Namespaces = [NS1];
  }
  Namespace NS1
  {
            Name = "lwc11";
            Classes = [CL1, CL2, CL3, CL4];
  }
  Class CL1
  {
            Name = "Model";
            Properties = [CL1PR1];
  }
  Property CL1PR1
  {
            Name = "PrimitiveTypes";
            Type = "List<Type>";
            IsAttribute = false;
            IsList = true;
            ValueType = "Type";
            MinimunCardinality = 0;
            MaximunCardinality = 2147483647;
            Cardinality = "0:*";
            Composition = false;
  }
  Property CL1PR2
  {
            Name = "Entities";
            Type = "List<Entity>";
            IsAttribute = false;
            IsList = true;
            ValueType = "Entity";
            MinimunCardinality = 0;
            MaximunCardinality = 2147483647;
            Cardinality = "0:*";
            Composition = false;
  }
  Class CL2
  {
            Name = "Type";
            Properties = [CL2PR1];
  }
  Property CL2PR1
  {
            Name = "Name";
            Type = "string";
            IsAttribute = true;
            IsList = false;
            MinimunCardinality = 1;
            MaximunCardinality = 1;
            Cardinality = "1:1";
            Composition = false;
  }
  Class CL3
  {
            Name = "Entity";
            Properties = [CL3PR1];
  }
  Property CL3PR1
  {
            Name = "Properties";
            Type = "List<Property>";
            IsAttribute = false;
            IsList = true;
            ValueType = "Property";
            MinimunCardinality = 0;
            MaximunCardinality = 2147483647;
            Cardinality = "0:*";
            Composition = false;
  }
  Class CL4
  {
            Name = "Property";
            Properties = [CL4PR1];
  }
  Property CL4PR1
  {
            Name = "Type";
            Type = "Type";
            IsAttribute = false;
```

```
                        IsList = false;
                        MinimunCardinality = 1;
                        MaximunCardinality = 1;
                        Cardinality = "1:1";
                        Composition = false;
            }
            Property CL4PR2
            {
                        Name = "Name";
                        Type = "string";
                        IsAttribute = true;
                        IsList = false;
                        MinimunCardinality = 1;
                        MaximunCardinality = 1;
                        Cardinality = "1:1";
                        Composition = false;
            }
}
```

Using this intrinsic meta-meta-model to essential (unchangeable) we can provide a DOM API to explore any metamodel as a model and consequently use it to produce more transformations, generating code or importers/exporters between metamodels.

3.2 Extensibility via Plugins and Reflection

This reflective capability opens the door to create generic projects able to manipulate and transform metamodels like models inside Essential.

An extension mechanism is built on top on Essential to allow an special type of Essential project to be installed in the tool as a plugin able to operate over metamodels.

As a sample, the tool comes with a sample plugin able to transform any metamodel to a set of Java POJOs classes. In order to test it, follow the steps:

1. Open the default project called **PluginMeta2Java**
2. Review its code to grasp its intented usage
3. In the menu select : `Plugin > Deploy Plugin`
4. Open the basic project from task 0.1 (**lwc11.essprj**)
5. Select the metamodel file from the Project Explorer.
6. Select the menu option: `Pluging > PluginMeta2Java`
7. Review the result in the output window:

```
Executing plugin PluginMeta2Java
  Executing transformation ConvertMetamodel()
   Serialize model to file: meta-java/model/model.ess
   Serialize plugin to file: meta-java/model/generator.ess
   foreach (Namespace ns in model.Namespaces)
    foreach (Class cl in ns.Classes)
     Apply Templates.genClass() > [meta-java/lwc11/Model.java]
     Apply Templates.genClass() > [meta-java/lwc11/Type.java]
     Apply Templates.genClass() > [meta-java/lwc11/Entity.java]
     Apply Templates.genClass() > [meta-java/lwc11/Property.java]
    foreach (Enum en in ns.Enums)
Transformed in 2420 ms.
```

A sample file generated like **meta-java/lwc11/Model.java** contains:

```
import java.lang;
package lwc11
{
  class Model
  {
            private List<Type> primitiveTypes;

            public List<Type> getPrimitiveTypes()
            {
                      return primitiveTypes;
            }
            public void setPrimitiveTypes(List<Type> value)
            {
                      primitiveTypes = value;
            }
}
```

The construction of a XML and XSD types, and MOF/XMI, ECORE model importers and exporters using this technique are under further investigation.

3.3 Reverser engineering

Creating components able to extract data from legacy code and/or databases schemas or Web services definitions and save such data in an Essential file opens the door to apply a Model-to-Model transformation chain able to extract and convert legacy code to be retargeted to a new platform.

This issue is a work in progress using Essential in the domain of semi-automating reverse engineering of Oracle Developer applications to model, and then back to RIA applications.

3.4 Betters editors for Essential

When models grow, the usability of the edition of models can be a problem. A text language is not always the better choice to each domain. To alleviate this, I used to complementing the approach using graphical editors when needed (for example, based Microsoft DSL Tools) able to generate a text version of the edited model in a well known Essential metamodel format.

The rest of the tool chain runs inside Essential. Same approach can be taken with GMF based editors in Java/Eclipse contexts.

In this case, humans deal with the graphics and the machine prefers the text based representations.

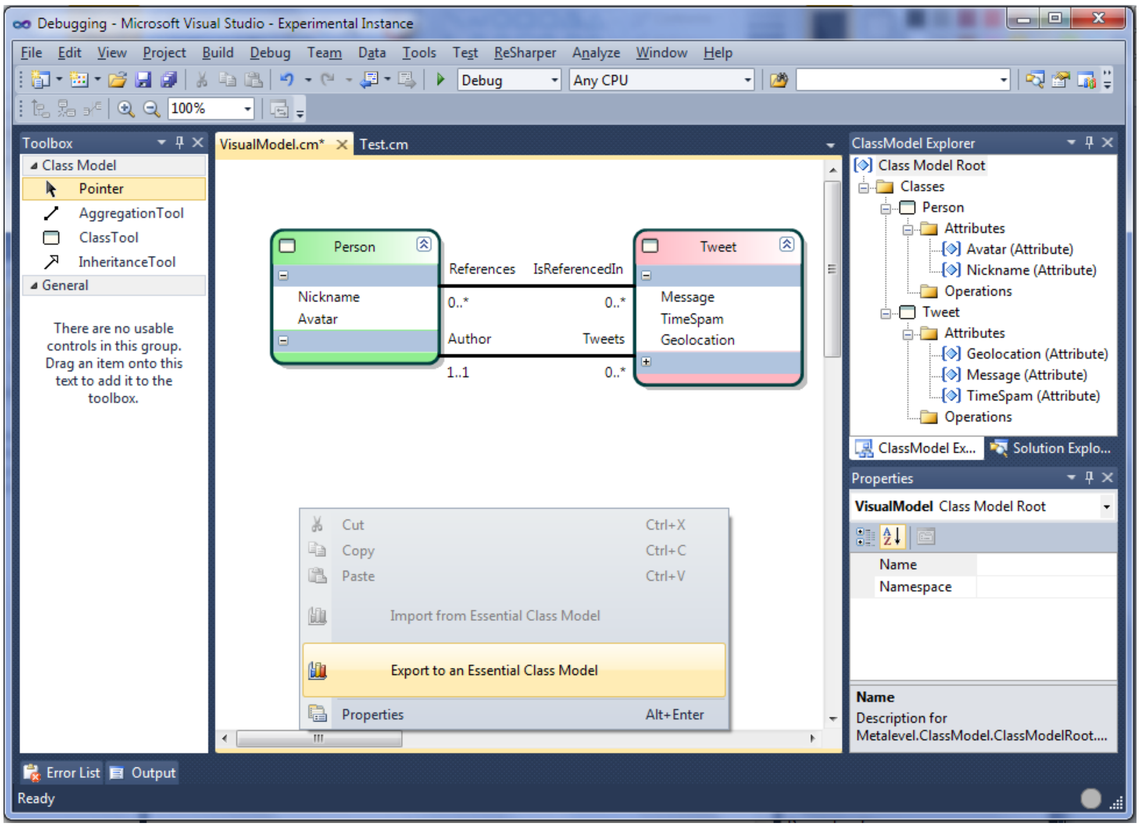


Figure 3. A visual class editor exporting essential models.

Sample Essential model file exported from the graphical editor:

```
using Meta.ClassModel;
namespace
{
           ClassModelRoot Root
           {
                      Name = "";
                      Classes = [Person, Tweet];
                      Relations = [PersonTweets, TweetQuotes];
           }
           Class Person
           {
                      Name = "Person";
                      UmlColor = UmlColor.Thing;
                      Properties = [
                                 Property Person_Nickname
                                 {
                                            Name = "Nickname";
                                            Type = "string";
                                            Size = 40;
                                 },
                                 Property Person_Avatar
                                 {
                                            Name = "Avatar";
                                            Type = "string";
                                            Size = 200;
                                 }
                      ];
           }
           Class Tweet
           {
           ...
  }
}
```