A Model-Driven Solution for Financial Data Representation Expressed in FIXML

Vahdat Abdelzad
University of Ottawa School of Electrical Engineering and Computer Science, Ottawa, Canada
v.abdelzad@uottawa.ca

Hamoud Aljamaan
University of Ottawa School of Electrical Engineering and Computer Science Ottawa, Canada
hjamaan@uottawa.ca

Opeyemi Adesina
University of Ottawa School of Electrical Engineering and Computer Science Ottawa, Canada
oades013@uottawa.ca

Miguel A. Garzon
University of Ottawa School of Electrical Engineering and Computer Science, Ottawa, Canada
mgarzon@uottawa.ca

Timothy C. Lethbridge
University of Ottawa School of Electrical Engineering and Computer Science, Ottawa, Canada
tcl@eecs.uottawa.ca

ABSTRACT
Accuracy of knowledge and information elicited via financial data processing is crucial to decision-making. In order to achieve this, we propose a solution based on the Umple modeling language for the Financial Information eXchange protocol (FIXML). The proposed solution includes syntactic and semantic analysis and automatic code generation developed in a test-driven approach. The solution also provides real-time visualization for FIXML documents. We then discuss our solution based on the following quality factors: development effort, modularity, complexity, accuracy, fault tolerance, and execution time. Finally, we applied our technique to the set of FIXML test cases defined in the FIXML case study, and we evaluated the results based on error detection and execution time.

Keywords
Model-Driven Development (MDD), Umple, FIXML, Automated Code Generation, Model Transformation.

1. INTRODUCTION
Accuracy of knowledge or information elicited via financial data processing is crucial to decision-making and prediction of investments and market trends by investors and portfolio managers in the financial domain [12]. Achieving this goal may be difficult or impossible without automated, dependable, flexible, and scalable implementation solutions for managing and processing huge volume of data emanating from daily market transactions. On this premise, the field of information processing has evolved with various approaches such as, data mining [7], and fuzzy logic [5] in order to reduce complexity experienced in processing high data volume. Virtually all approaches to process and gain knowledge for decision-making require or depend on software-controlled systems. Model-based design and automated code generation (or auto-coding) methods [6, 11], thereby provides inter-connected partial solutions to developing these systems with minimum effort and defects. Proponents of these methods [4, 6, 8], argue that they tend to deliver quality artifacts; because of their promises of higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors.

This paper provides a transformation solution to an electronically financial transactions expressed FIXML format. Our transformation approach reverse engineers FIXML data into Umple model, which then translates into targeted programming languages (e.g. Java). In this transformation, according to OMG levels Umple is seen as M1 level with Umple classes representing FIXML schema. Umple [2, 3] is an open-source code generation and modeling tool we have adopted for FIXML [17] transformation. Our choice of Umple is based on its strengths and philosophies. Firstly, the lightweight capabilities of Umple allow modelers and programmers to seamlessly build applications [2] by embedding code within the textual model, which is impossible with traditional solutions. Secondly, according to [10] Umple has been developed with a focus on three key qualities: usability, completeness, and scalability. Usability has been considered key, because we want to facilitate rapid modeling with fewer defects. Moreover, it is important to be able to model systems of arbitrary size and manage models without slowing down. These are prerequisite to any successful tool for generating code from a plethora of data, which is usually generated, and often require processing from the financial domain. Thirdly, the integration of FIXML to Umple only requires us to define a grammar to parse instances of its meta-model. Fourthly, Umple cannot only generate Java and C++, as the solution demanded, but also SQL, PHP and Ruby from the textual representation of FIXML data. Umple achieved these benefits; since it was written in itself, which gives it ability to construct automatically internal model representation of the input text.

Our solution allows input FIXML text to be processed in all its development environments, including UmpleOnline [16], its Eclipse plugin, or its command-line tool. Umple's parser analyses the input text statically against the defined FIXML grammar. Upon successful static analysis, Umple constructs the internal model of the input text as an instance of Umple's own metamodel. This is then used to generate the target languages. The results obtained from the test cases have shown that the code generated is syntactically and semantically accurate, and also robust, discovering invalid inputs in given test cases #3, #4, #7, #8 [14].

We corrected these cases and presented the time taken to generate actual code from all the given test cases.

The rest of this paper is organized thus: in Section 2, we present more detailed information about Umple. In Section 3, 4, and 5, we present detailed information about our solution, results and evaluation, and conclusions respectively.

2. UMPLE

Umple allows textual modeling in UML and can be seen as both a modeling and a programming language. Umple allows you to specify elements such as the following:

- Classes and Interfaces
- Associations: Umple supports multiplicity constraints and manages referential integrity.
- Attributes: Can be constrained in various ways.
- State Machines: Transitions, entry/exit actions, nested and concurrent states, and do activities.
- Aspect Orientation: Code that can be run before or after Umple-defined actions on attributes, associations, and the elements of state machines.
- Tracing: Sublanguage of Umple. It allows developers to specify tracing at the model level.
- Patterns such as singleton and immutable.

Umple generates code in Java, PHP, C++, Ruby, and SQL. It also generates API documentation, metrics and various diagram types.

The Umple team has formulated a number of philosophies that direct its research vision [1].

The first philosophy is that Umple sees modeling as programming and vice versa. With Umple, UML can be expressed textually and so a modeler can see UML visually and textually, while a programmer can see UML coded abstractly.

The second philosophy is that there is no need for round tripping (i.e. editing generated code), since any special-purpose code can be embedded in Umple as necessary. The third philosophy is that usage of Umple can start from an existing system and UML constructs can be added incrementally. Hence, Umple will parse programming languages code as part of Umple code. The fourth philosophy states that Umple goes beyond UML boundaries; for instance Umple directly implements patterns and other common programming idioms.

The fifth philosophy states that base language code added to an Umple program corresponds to UML's concept of an action language. Development in Umple can take a bottom up approach starting with code, and add UML constructs as he gains confidence, or a top-down approach in which a developer can start by writing UML constructs in Umple and then iteratively add code for algorithmic operations.

We provide developers with three types of tools to develop systems using Umple:

- **An Eclipse plugin**. This gives developers the full power of the Eclipse environment as they use Umple.
- **UmpleOnline**. This is an interactive website [16] that allows anyone to instantly experiment with Umple on the web. It has two panels; one for Umple textual code and another for visualizing UML constructs. The user can explore examples or create his or her own and save them in the cloud. UML diagrams are generated as the user types.
- **Command-line based compiler**. This allows Umple developers to compile their Umple systems from command line.

The tools and language to implement Umple are built using a test-driven approach. This approach enabled our team to quickly develop a functional version of the language, without hindering future development or features, and without breaking other aspects of Umple. Test Driven Development (TDD) enables the software to evolve based on feedback received from early adopters, and has enabled our team to use early versions of the Umple language to develop and enhance future versions. The Umple toolset and language, which were originally written in Java, were long ago fully rewritten in Umple, and is now developed and maintained in Umple itself.

The Umple internal components include: a parser, an analyzer that generates an instance of Umple's metamodel from the parse tree, synchronization engine (to allow diagrams to be edited and the resulting changes being applied to the text) as well as several code generators and model-to-model transformation engines.

The Umple testing process in Figure 1 is capable of testing all artifacts within the scope of Umple. In other words, we test Umple as well as representative systems created using Umple.

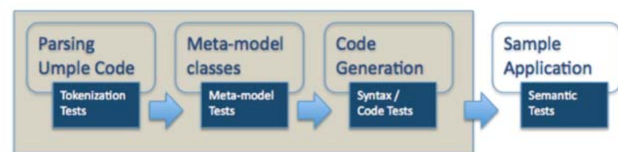

Figure 1. Umple Components

At present, there are over 3280 tests that span all components of the Umple infrastructure. Testing the Umple Parser is centered on the tokenization of Umple code. The tests in this area ensure that Umple models parsed and tokenized as we expect. Testing the metamodel classes ensures that Umple will be able to maintain valid internal representations of a model. Testing the code generators ensures that we generate valid base language code (i.e. Java, C++, PHP, etc.). This is done by comparing the expected code versus the actual generated code. Here, we are testing that the syntactic translation of the Umple metamodel instance into the generated base language is correct. Finally, we use a test bed of Umple code to test whether that code behaves as expected.

3. OUR SOLUTION TO THE FIXML CHALLENGE

To solve the problem presented in this challenge, we define an extension to the Umple grammar to parse FIXML documents, and

process them such that they become instances of Umple's own internal metamodel.

We use Umple's mixin capability to inject algorithms for analysis of the FIXML input into Umple's semantic analyzer. The mixin capability helps us not to alter base Umple code but allows us to create the FIXML extension as a separate concern. The Umple mixin mechanism automatically adds the algorithms to the core of Umple.

The first critical step in our process is to create valid models from FIXML documents. To achieve this, we need to perform syntactic and sematic validation of FIXML documents. In order to support this, we validate FIXML document in two phases. In the first phase, our parser verifies that we have a syntactically valid FIXML document; it produces an internal syntax tree but does not cover semantic checking yet. In the second phase, we do semantic checking for FIXML documents. This validates that we have the same opening and ending tag names, for example. In the second step of having a valid model, we get help from Umple meta-model which bring us semantic constraints in order to have a valid model and also generate completely valid code for target programming languages.

For syntactic validation, we have defined a set of grammars to parse FIXML documents. The FIXML grammar is shown in Listing 1. Umple has its own EBNF syntax [13] which has special features adapted to processing source that contains multiple languages.

Listing 1. Umple Grammar for FIXML

```
fixml: [[ fixmlDefinition ]] |  [[ fixmComment ]]  |  [[ fixmlDoc ]]
fixmComment: <?xml [[tagDefinition]]* ?>
fixmlDoc: <! [**value] >
fixmlDefinition: <FIXML > [[ fixmlContent ]]* </FIXML>
fixmlContent: [[endContent]] | [[ extendContent ]]
endContent: < [~name]   ( [[ tagDefinition ]])* />
extendContent: < [~name] ( [[tagDefinition]] )* > (
    [[endContent]] | [[extendContent]] | [[attContent]])* < ( / )
    [~name] >
tagDefinition: [name] = "[**value]"
attContent: < [~name] > [**value:\<] < ( / ) [~name] >
```

In Listing 1, the rule name "fixmlDefinition" is composed of a symbol "<FIXML>", followed by a non-terminal called "fixmlContent", then the symbol "</FIXML>". This rule is the main rule which Umple's parser detects FIXML documents and parses them. We use single square brackets ("[" and "]") to match various types of tokens such as identifiers, and double square brackets ("[[" and "]]") for rule-based non- terminals. Rules names are added to the tokenization sequence. Symbols (e.g. terminals), such as "<FIXML>" are used in the analysis phase of the parsing (to decide which parsing rule to invoke), but they are not added to the resulting tokenization string for later processing.

Our grammar syntax allows for rapid language creation. The language authors do not need to worry about the complex, repetitive and error prone regular expressions used to define common structures such as string sequences, decimal numbers, alphanumeric strings, and arbitrary code blocks as would be required when using other parsers like Antlr [1].

In our solution, we consider tag attributes to be Umple attributes for our model. In the process of analysis, we detect the type of attributes (Integer, Double, and String) and use the correct Umple types for those attributes. This brings us correct and robust model and code generation. By using this capability, we can detect the majority of mistakes in the values of attributes. Moreover, we automatically create related *set* and *get* methods for those attributes. Indeed, we define attributes with private visibility and implement automatically related *set* and *get* methods so as to support data encapsulation. For example, Listing 2 shows a FIXML document in which there is a tag with three attributes. According to the values of attributes, we have two integer attributes and one float attribute. The generated code for the FIXML document in Listing 2 is represented in Listing 3. We removed *set* and *get* methods and other codes because of space limitation. All generated code can be obtained online through UmpleOnline [16].

Listing 2. A sample FIXML document

```
<FIXML>
    <Order ClOrdID="123456" Side="2" Px="93.25">
    </Order>
</FIXML>
```

Listing 3. Java code with proper attribute types

```
class Order{
  private int ClOrdID;
  private int Side;
  private double Px;
  //The rest of code
}
```

In the proposed solution in [9], Lano et al. used an instance variable in generated code for every nested tag in FIXML documents. This approach is also applied to the nested tags with the same name (which results in the same objects). Listing 4, for example, shows three nested tags with the same name called Pty. The generated code for Java based on the solution proposed in [9] is shown in Listing 5. In Listing 5, we can see that there are three instance variables and a constructor with three parameters. This approach is not correct for large FIXML documents and also it doesn't have a good code implementation for associations in model-driven development. In fact, when we have a large FIXML document with a tag which has more than 255 nested tags, this approach will not work. According to the solution in [9], we should add all of those object instances as parameters to the related class constructors. However, it is impossible because we have a limitation in the number of parameters in programming languages (e.g. limitation of 255 words for method parameters in Java). We have tackled the issues with the concepts of association in the model and arrays as inputs for those same objects in the

implementation. Listing 6 shows our generated code in which we have just an instance variable and a constructor with a parameter. This removes the limitation related to the number of parameters in programming languages. On the other hand, we have just an instance variable which helps us not to lose the model-driven meaning of associations even in the generated code. It means that we have an instance variable for each association without worries about multiplicity.

Listing 4. Umple Grammar for FIXML

```
<PosRpt>
  <Pty ID="OCC" R="21"/>
  <Pty ID="99999" R="4"/>
  <Pty ID="C" R="38"/>
</PosRpt>
```

Listing 5. Java code generated by the solution in [9]

```
class PosRpt {
  Pty Pty_object_1 = new Pty("OCC","21");
  Pty Pty_object_2 = new Pty("99999","4");
  Pty Pty_object_2 = new Pty("C","38");
  PosRpt (Pty Pty_1, Pty Pty_2, Pty Pty_3){
      this.Pty_object_1 = Pty_1;
      this.Pty_object_2 = Pty_2;
      this.Pty_object_3 = Pty_3;
  }
  PosRpt (){
  }
}
```

Listing 6. Java Code generated by our proposal

```
class PosRpt{
  private List<Pty> Pty_Object;
  public PosRpt(Pty... allPty_Object)
  {
    Pty_Object = new ArrayList<Pty>();
    boolean didAddPty_Object = setPty_Object(allPty_Object);
  }
  public PosRpt()
  {
    Pty_Object.add(new Pty("OCC", 21));
    Pty_Object.add(new Pty("99999", 4));
    Pty_Object.add(new Pty("C", 38));
  }
}
```

4. RESULTS AND EVALUATION

In this section, we present the results and evaluation of our implementation solution based on the following parameters: development effort, modularity, complexity, accuracy, fault tolerance, and execution time.

4.1 Development Effort, Modularity, and Complexity

In the design and implementation of our solution, we raised the level of abstraction, and minimized development time as well as complexity for future changes to a considerable level amount of time. To achieve these qualities, we defined a simple grammar for parsing FIXML documents. Umple uses the defined grammar for automatic construction of a parse tree representing the input text and generates a model that is independent of any target language. We achieved this with minimum effort and belief that future extension or modification will require minimum effort too.

4.2 Accuracy

The code generated from any given FIXML text, in every target language supported by our solution, conforms to their native syntax and semantics. We achieved syntactic conformance by invoking static analyzer embedded in Umple compiler. . With this approach we were able to uncover errors and modify our implementation to certify syntactic correctness of the generated code. In the same vein, we have adopted the concept of associations in order to preserve semantics as expected. With Umple, creation of links by associations ensures that unique names are created for every instance variables of the same class and preserves the underlying semantics.

4.3 Fault Tolerance

Our solution is robust and detected malformed FIXML documents provided as the test cases. The solution parses all the test cases available at [14], after some modification to some of the test cases. The solution we developed parses test cases #1, #2, #5 and #6 without modification. However, the remaining set of test cases requires some modification.

Firstly, test case #3 failed because the <Order> tag was closed with “<Order>” tag instead of “</Order>”. Secondly, test case #4 failed because version and unicode values were quoted with single quotes. We made modification by changing the quotes from single to double. Thirdly, test case #7 failed because the <Order/> tag was given, instead of “<Order>”. Lastly, test case #8 failed for the following reasons. The tag “<FIXML>” was not closed with the corresponding “</FIXML>” tag. Its “<Order>” tag was not closed with the corresponding “</Order>”. There was no matching tag for the corresponding “</OrderMessage>” tag. The “<Hdr>” tag was closed with an “<Hdr>” tag, instead of “</Hdr>”. We corrected these malformed tags In order to verify our solution, you may visit [15].

4.4 Execution Time

We have instrumented our compiler with a Timer to measure the time taken to process an input file and produce the target source code. More specifically, the Timer measures the time taken to 1) parses an input file, 2) to analyze and build an instance of the Umple metamodel and 3) to generate code which involves creating a file (.Java, .C++, etc.).

Table 1 summarizes the executions times in milliseconds, for each of the eight FIXML test cases [14]. The executions times have been split to reflect the three main stages of the transformation process: parsing the FIXML code, analyzing the tokens to build an instance of the Umple metamodel, and generating Java code (one of our target languages). The tests were executed on a machine exhibiting the following characteristics:

- Intel Core i5-2400 CPU @ 3.10GHz
- RAM: 8.00 GB
- Windows 8 - 64 bits

As presented in the table, the parsing and analyzing times are constant for most of the cases, showing that our technique gives good performance results even for larger inputs, as is the case for the test #8. The code generation stage results depend on the size of the file generated (Java files in this case) and this explains the variations in the execution times.

5. DEMONSTRATION

As mentioned in Section 3 of this paper, it is possible to create an Umple model using one of our three tools: the Eclipse plugin, the command-line based compiler or the web-based application named UmpleOnline [16]. The quickest way to compile and generate code with Umple is to go to UmpleOnline, and copy-paste one of the eight FIXML test cases [14] into the code editor (left-pane). As shown in Figure 2 you can visualize the corresponding UML class diagram with attributes and associations between objects (right pane) and/or generate code. At this moment, Umple supports code generation in Java, C++, PHP, Ruby, Ecore and SQL (to create your database tables based on your model). Umple not only gives you a high-quality code implementation but also a way to better visualize your models.

To use the command-line tool, you can download it at http://dl.umple.org, and run the command: "*java –jar umple.jar YourTestcase.ump"*. The command-line tools and the Eclipse plugin process files that conventionally have the extension .ump

6. CONCLUSION

In this paper, we proposed and implemented a solution for automatic object-oriented code generation for financial data representation expressed in FIXML. In order to achieve this, we utilized Umple, which includes mechanisms for parsing, analysis, and automatic code generation. Extending the Umple grammar to support FIXML satisfied the requirement for accurate syntactic processing of FIXML documents and also provides a flexible path for ongoing modification. Umple automatic code generation supports several programming languages and other software artifacts. Our solution also provides a real-time visualization for FIXML documents without code generation. This visualization includes UML class diagrams showing classes, attributes, and associations and inheritance relationships between those classes.

7. REFERENCES

[1] Antlr Technology: 2014. *http://www.antlr.org/*.

[2] Badreddin, O. 2010. Umple: a model-oriented programming language. *2010 ACM/IEEE 32nd International Conference on Software Engineering*. 2, (2010), 337–338.

[3] Badreddin, O., Forward, A. and Lethbridge, T.C. 2012. Model oriented programming: an empirical study of comprehension. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research* (2012), 73–86.

[4] Czarnecki, K. and Eisenecker, U. 2000. *Generative Programming: Methods, Tools, and Application*. Addison-Wesley.

[5] Daniel, S. and Tettamanzi, A.G.B. 2006. Reasoning and Quantification in Fuzzy Description Logics. *Fuzzy Logic and Applications, 6th International Workshop, WILF* (2006), 81–88.

[6] Denney, E. and Fischer, B. 2009. Generating Code Review Documentation for Auto-Generated Mission-Critical Software. *Third IEEE International Conference on Space Mission Challenges for Information Technology* (Jul. 2009), 394–401.

[7] Grossman, R. and Gu, Y. 2008. Data mining using high performance data clouds. *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08* (New York, New York, USA, Aug. 2008), 920.

[8] Kleppe, A., Warmer, J. and Bast, W. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

[9] Lano, K., Yassipour-Tehrani, S. and Maroukian, K. Case study: FIXML to Java, C# and C++. *Transformation Tool Contest - ttc2014*.

[10] Lethbridge, T.C. 2013. Key Properties for Comparing Modeling Languages and Tools : Usability , Completeness and Scalability. *Comparing Modeling Approaches at MODELS 2013* (2013).

[11] Nakićenović, M.B. 2012. An Agile Driven Architecture Modernization to a Model-Driven Development Solution-An industrial experience report. *International Journal On Advances in Software*. 5, 3, 4 (2012), 308–322.

[12] O'Brien, J. 1970. How market theory can help investors set goals, select investment managers and appraise investment performance. *Financial Analysts Journal*. 26, 4 (1970), 91–103.

[13] Syntactic metalanguage -- Extended BNF Standard: 2014. *http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153*.

[14] Transformation Tool Contest - ttc2014-fixml GitHub: 2014. *https://github.com/TransformationToolContest/ttc2014-fixml/tree/master/test_cases*.

[15] Umple home page: 2014. *http://www.umple.org*.

[16] UmpleOnline: 2014. *http://cruise.eecs.uottawa.ca/umpleonline/*.

[17] 2012. FIXML 4.4 Schema Version Guide.

Table 1. Execution time for the eight Fixml test cases

	Execution Time (in ms)							
Component	**Case #1**	**Case #2**	**Case #3**	**Case #4**	**Case #5**	**Case #6**	**Case #7**	**Case #8**
Parsing	314	333	324	331	396	607	322	329
Analyzing	17	20	18	20	27	41	17	18
Generating Java Code	198	430	265	294	1543	3572	221	214
Total Time:	529	783	607	645	1966	4220	560	561

Draw on the right, write (Umple) model code on the left. Generate Java, C++, PHP or Ruby code from your models.
Visit the User Manual or the Umple Home Page for help. Download Umple Report an Issue

Line=1 Create Bookmarkable URL

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <FIXML>
3  <PosRpt RptID="541386431" Rslt="0"
4          BizDt="2003-09-10T00:00:00" Acct="1"
   AcctTyp="1"
5          SetPx="0.00" SetPxTyp="1" PriSetPx="0.00"
   ReqTyp="0" Ccy="USD">
6        <Hdr Snt="2001-12-17T09:30:47-05:00"
   PosDup="N" PosRsnd="N" SeqNum="1002">
7  <Sndr ID="String" Sub="String" Loc="String"/>
8  <Tgt ID="String" Sub="String" Loc="String"/>
9  <OnBhlfOf ID="String" Sub="String" Loc="String"/>
10 <DlvrTo ID="String" Sub="String" Loc="String"/>
11 </Hdr>
12 <Pty ID="OCC" R="21"/>
13 <Pty ID="99999" R="4"/>
14 <Pty ID="C" R="38">
15 <Sub ID="ZZZ" Typ="2"/>
16 </Pty>
17 <Qty Typ="SOD" Long="35" Short="0"/>
18 <Qty Typ="FIN" Long="20" Short="10"/>
19 <Qty Typ="IAS" Long="10"/>
20 <Amt Typ="FMTM" Amt="0.00"/>
21 <Instrmt Sym="AOL" ID="KW" IDSrc="J"
   CFI="OCASPS" MMY="20031122"
22                Mat="2003-11-22T00:00:00"
   Strk="47.50" StrkCcy="USD" Mult="100"/>
23 </PosRpt>
24 </FIXML>
```

SAVE & RESET
TOOLS
Select Example
Choose from Dropbox
DRAW
Class [c]
Association [a]
Generalization [g]
Delete [DEL]
Undo [ctrl+z]
Redo [ctrl+y]
Sync Diagram
GENERATE
Java Code
Generate Code
OPTIONS

OnBhlfOf: ID : String, Sub : String, Loc : String
DlvrTo: ID : String, Sub : String, Loc : String
Sub: ID : String, Typ : Integer
Hdr: Snt : String, PosDup : String, PosRsnd : String, SeqNum : Integer
Pty: ID : String, R : Integer
Qty: Typ : String, Long : Integer, Short : Integer
PosRpt: RptID : Integer, Rslt : Integer, BizDt : String, Acct : Integer, AcctTyp : Integer, SetPx : Double
Tgt_Object, 1 OnBhlfOf_Object, DlvrTo_Object, 1 Sub_Object, 1 Hdr_Object, 1 Pty_Object, 3 Pty_Object, 1 Qty_Object, 1 PosRpt_Object

Figure 2. Test case #2 [14] **loaded in UmpleOnline – http://try.umple.org**