Exploring a Model-Oriented and Executable Syntax for UML Attributes

Omar Badreddin, Andrew Forward, Timothy C. Lethbridge

Abstract Implementing UML attributes directly in an object-oriented language may not appear to be complex, since such languages already support member variables. The distinction arises when considering the differences between modelling a class and implementing it. In addition to representing attributes, member variables can also represent association ends and internal data including counters, caching, or sharing of local data. Attributes in models also support additional characteristics such as being unique, immutable, or subject to lazy instantiation. In this paper we present modeling characteristics of attributes from first principles and investigate how attributes are handled in several open-source systems. We look code-generation of attributes by various UML tools. Finally, we present our own Umple language along with its code generation patterns for attributes, using Java as the target language.

Key words Attributes, UML, Model Driven Design, Code Generation, Umple, Model-Oriented Programming Language.

1 Introduction

A UML attribute is a simple property of an object. For example, a Student object might have a *studentNumber* and a *name*. Attributes should be contrasted with associations (and association ends), which represent relationships among objects.

Constraints can be applied to attributes; for example, they can be immutable or have a limited range. In translating UML attributes into languages like Java it is common to generate accessor (get and set) methods to manage access.

Omar Badreddin, Andrew Forward, Timothy C. Lethbridge
School of Electrical Engineering and Computer Science, University of Ottawa, Canada K1N 6N5
e-mail: obadr024@eecs.uottawa.ca, aforward@eecs.uottawa.ca, tcl@eecs.uottawa.ca

In this paper, we study the use of attributes in several systems and discuss how to represent attributes in a model-oriented language called Umple. Umple allows models to be described textually as an extension to Java, PHP, Ruby or C++. We present code-generation patterns for attributes as used by Umple for the Java language.

1.1 A Quick Look at Umple

Umple is a set of extensions to existing object-oriented languages that provides a concrete syntax for UML abstractions like attributes, associations, state machines. To distinguish between Umple and Java, the Umple examples use dashed borders in shading, and Java examples use solid-line borders with no shading.

Figure 1 is a snippet of Umple on the left, with its corresponding UML diagram on the right. Methods have been left out of this example; this illustrates one of the features of Umple, the ability to use it incrementally, first to create high level models, and later on to add more and more implementation detail until the system is complete.

```
class Student {}
class CourseSection {}
class Registration {
  String grade;
  * -- 1 Student;
  * -- 1 CourseSection;
}
```

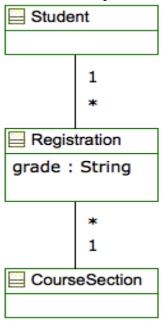

Fig 1. Umple class diagram for part of the student registration system

Figure 1 shows two associations and an attribute so that the reader can see how they are defined in Umple. The remainder of the paper focuses exclusively on attributes.

One of our motivations is our previous research [1] indicating that most developers remain steadfastly code-centric; hence visual modeling tools are not being adopted as widely as might be desired. Another motivation is that there is much repetitive code in object-oriented programs. We wish to incorporate abstractions to promote understandability and reduce code volume [2].

An Umple program contains algorithmic methods that look the same as their Java counterparts. Constructors, instance variables and code for manipulating attributes, associations, and state machines are generated.

Umple is intended to be simple from the programmer's perspective because there is less code to write and there are fewer degrees of freedom than in Java or UML. Despite the restrictions in Umple, it is designed to have ample power to program most kinds of object-oriented systems. The current version of Umple is written in itself.

Please refer to [3] for full details about Umple. The Umple language can be explored in an online editor [3], which includes many examples.

2. Attributes in Practice: A Study of Seven Systems

To ground our work in the pragmatics of industrial software development, we analyzed how real projects implement attributes. This will help us identify code-generation patterns and areas where Umple could be improved.

Key goals of our empirical analysis of software attributes are to determine how attributes are defined, accessed and used in practice, and also to find attribute patterns that can enhance the vocabulary with which attributes are defined in Umple

For our research, we considered seven open-source software projects. The criteria by which the projects were selected are described below, followed by a review of the results and the implications for building a model-oriented syntax to describe attributes. We sampled existing software systems by selecting a random sample of projects from selected repositories. A candidate repository contained at least 1000 full projects in Java or C#. We analyzed 33 repositories, and selected three that met our criteria

Candidate projects were selected by randomly picking a repository, then randomly selecting a language (Java, or C#), and finally randomly selecting one of the first 1000 projects. The 7 projects analysed include: from GoogleCode: fizzbuzz, ExcelLibrary, ndependencyinjection and Java Bug Reporting Tool; from SourceForge: jEdit and Freemaker; and from Freecode (formerly Freshmeat): Java Financial Library.

We documented all member variables. For each we recorded the project, namespace, object type, variable name, and other characteristics presented in Table 1.

2.1 Analysis and results

We used reverse engineering tools to extract member variables from source code, and manually inspected each attribute. We identified 1831 member variables in 469 classes. Of the member variables identified, 620 were static (class variables) and 1211 were instance variables. Table 2 gives a distribution of the types of static variables.

Table 1 Categorizing member variables.

Category	Values	Description
Set in Constructor	No, Yes	Is the member variable set in the object's constructor?
Set Method	None, Simple, Custom	Is the variable public, or does it have a setter method? If so, is there custom behavior apart from setting the variable (such as validating constraints, managing a cache or filtering the input)
Get Method	None, Simple, Custom	Is the variable public, or does the variable have a getter method? If so, does it have any custom behavior like returning defaulted values, using cached values or filtering the output.
Notes	Free Text	Other characteristics such as whether the variable is static, read-only, or derived.

Table 2. Distribution of static (class) variables.

Object Type	Frequency	%	Description
Integer	431	69%	All whole number types including primitive integers, unsigned, and signed numbers.
String	53	9%	All string and string builder objects.
Boolean	29	5%	All True/False object types.
Other	107	17%	All other object and data types
Total	620	100%	

Out of the 620 static members analyzed, 90% were read-only constants, 69% were publically visible, and 83% were written in ALL_CAPS, a common style for static variables. From this point onwards, we will focus on the instance variables.

Table 3 gives the distribution of all instance members (i.e. non-static variables) for the five basic attribute types. The 'other' includes custom data types, plus types corresponding to classes like *Address*. Member variables consist of attributes, associations and internal data. To help determine which variables are most likely attributes; we used a two-phased approach. First, we analyzed whether the variables were included in the object's constructor and whether the member variable had get and set accessor methods. This analysis is shown in **Table 4**.

Only 3% of the variables were initialized during construction, could be overwritten in set method, and accessed in a get method. The most common occurrence was no access to a variable at all (not in constructor, and also no accessor methods). The second most common was a variable whose value was set only after construction.

To filter out potential internal data (local variables), we removed from our potential attributes list all variables that did not have get. We also visually inspected the list and observed that most no-getter variables were cached objects and results (i.e. size or length), or user-interface controls. In total, 637 member variables were removed during this process. We also filtered out five member variables with the word *cache*, or *internal* in their name; as they most likely also refer to internal data.

Table 3. Distribution of instance variable types.

Object Type	# of Variables	%	Description
Integer	326	27%	All whole number types including primitive integers, unsigned, and signed numbers.
String	169	14%	All string and string builder objects.
Boolean	121	10%	All True/False object types.
Double	12	1%	All decimal object types like doubles, and floats
Date / Time	9	1%	All date, time, calendar object types.
Other	574	47%	All other data types
Total	1211	100%	

To find variables representing attributes, as opposed to associations, we worked recursively. An attribute is considered to have as its type either: a) a simple data type identified in the first five rows of Table 3, or b) a class that only itself contains instance variables meeting conditions a and b, with the proviso that in this recursive search process, if a cycle is found, then the variable is deemed an association. This approach was partially automated (identifying and removing 12 association member variables) where both ends of the association were defined within the system. The remaining variables were inspected manually, and subjective judgments were made to categorize the variable type as entity or complex. An entity class is one that is comprised of only primitive data types, or associations to other entity classes. A complex class is comprised of primitive data, as well as associations to other complex classes. **Table 5** was used to help distinguish class categories.

Table 4. Analyzing all instance variables for presence in the constructor and get/set methods**.**

Constructor	Setter	Getter	Freq	%	Likelihood of being an attribute (High, Medium, Low)
Yes	Yes	Yes	32	3%	High, full variable access
Yes	Yes	No	8	1%	Low, no access to variable
Yes	No	Yes	44	4%	High, potential immutable variable
Yes	No	No	160	13%	Low, more likely an internal configuration
No	Yes	Yes	318	26%	High, postpone setting variable
No	Yes	No	41	3%	Low, no access to variable
No	No	Yes	179	15%	Medium, no access to set the variable
No	No	No	429	35%	Low, no access at all to set/get variable
Total	1211	100%			

Table 5. Entity versus complex object type criteria hinds**.**

Entity Class	Complex Class
Properties, Formats, Types and Data	Nodes, Worksheets
Files, Records, and Directories	Writers, Readers
Colors, Fonts, and Measurements	Engines, Factories and Strategies
Indices, Offsets, Keys and Names	Proxies, Wrappers, and Generic Objects
	Actions, Listeners, and Handlers
	Views, Panes and Containers

This process identified internal, attribute and association variables. Once complete, we were left with 457 potential attributes. The distribution of attribute types is shown in Table 6. As expected, most potential attributes are integers, strings and Booleans.

Table 6. Distribution of attribute types.

Object Type	Freq.	%	Description
Integer	200	44%	All whole number types (e.g. integers, signed, and unsigned).
String	102	22%	All string and string builder objects.
Boolean	67	15%	All True/False object types.
Double	6	1%	All decimal object types like doubles, and floats
Date / Time	5	1%	All date, time, calendar object types.
Other	77	17%	All other data types
Total	457	100%	

Table 7 divides attributes into 4 categories. Only 29 attributes (6%) had immutable-like qualities (available in the constructor, with no setter). About 31% of the attributes were managed internally with no setter and not available in the constructor. Finally, only about 11% of the attributes were available in the object's constructor.

Table 7. Constructor and Access Method Patterns (all attributes have a get method).

Constructor	Setter	Frequency	%	Probable Intention
Yes	Yes	23	5%	Fully editable
Yes	No	29	6%	Immutable
No	Yes	262	57%	Lazy / postponed initialization
No	No	143	31%	Derived or calculated attribute
Total	457	100%		

Implementation of set and get methods: As described in Table 1, a set or get method, if present, can be simple or custom. Table 8 illustrates the frequency of the various combinations of attribute set and get methods.

Table 8. Distribution of attribute properties based on type of setters and getters.

Setter	Getter	Frequency	%
Simple	Simple	250	55%
Simple	Custom	1	0%
Custom	Simple	9	2%
Custom	Custom	25	5%
None	Simple	46	10%
None	Custom	126	28%
Total	457	100%	

Over 55% of attributes had simple set / get mechanisms, 10% had simple get methods with no set method, and the remaining 35% had at least some custom set or get method.

Attribute Multiplicities: We distinguished between *one* (0..1 or 1) and *many* (*) based on the attribute type. List structures and object types with a plural noun (e.g. *Properties*) were identified as *many*, all other structures were identified as *one*.

Overall 93% of attributes had a multiplicity of *one*, leaving only 7% with a *many* multiplicity. To more finely categorize the multiplicity types would be too subjective, as the multiplicity constraints are programmed in diverse ways.

Characteristics of custom access methods: The following custom set method implementations were observed: having a caching mechanism, lazy loading, updating multiple member variables at once, and deriving the underlying member variable's value based on the provided input.

The following custom *get* method implementations were observed: constant values returned, default values returned if the attribute had not been set yet, lazy loading of attribute data, attribute values derived from other member variable(s), and the attribute value returned from a previously cached value. A summary of the implementation types for set and get methods is in Table 9.

Table 9. Distribution Set and Get Method implementations.

Method Implementation	Description	Freq.	%
Derived Set	Input filtered prior to setting variable's value	4	1%
Other Custom Set	Caching / updating multiple members at once	30	7%
Derived Get	Based on a cache, or other member variables	105	23%

Other Custom Get	Custom constraints applied to variable	28	6%
Constant Get	Always returns the same value	19	4%

The frequencies in Table 9 are based on the total number of attributes and not simply those attributes with custom set or get methods. The most interesting observation from this table is that almost a quarter of all attributes were somehow derived from other data of the class.

2.2 Key findings

Key findings based on the results above include

- **Simple set and get methods**: Many attributes follow a simple member variable get and set approach, suggesting that such behavior could be the default, helping to reduce the need for explicit setters and getters.
- **Immutable attributes**: Few attributes are set during construction, implying a separation between building objects and populating their attributes. Despite this, we believe it is still important to allow attributes to be immutable and, hence, set only in the constructor. Immutability helps ensure the proper implementation of hash codes and equality; for example, to allow consistent storage and retrieval from hash tables. It is also important for asynchronous and distributed processing where tests need to be done to see if one object is the same as another.
- **Attribute multiplicities**: Attribute multiplicities are almost always 'one' (93%). Based on this, Umple only supports the generic 'many' multiplicity and not specific numeric multiplicities as found in associations.
- **Static attributes**: Class level attributes (i.e. static) were mostly written in ALL_CAPS (83%); a convention that could be added directly to a language, removing the need for the 'static' keyword.

By analyzing existing projects we were able to align our model-oriented language Umple with the observed trends in representative software projects. This alignment will be expanded upon in the next section. We were also able to provide code generation that is aligned to industry practice – in order to help make the quality of the generated code similar in style and quality to code that a software developer would write him or herself.

3 Umple Syntax for Attributes

In this section we show how the Umple language (introduced in Section 1) allows the programmer to specify attributes, with common characteristics found in practice as presented in the last section. In UML, attributes represent a special subset of semantics of UML associations, although pragmatically we have found it more useful in Umple to consider them as entirely separate entities.

The main features of Umple's syntax for attributes, and its code generation, result from answering the following three questions.

- Q1: Is the attribute value required upon construction?
- Q2: Can the attribute value change throughout the lifecycle of the object?
- Q3: What traits / constraints limit the value and accessibility of the attribute?

As we discuss in Section 4, most current code generators provide the most liberal answers to the questions above: no, the value is not required upon construction, yes the attribute value can change, and no there are no constraints on or special traits of the attribute. In UML, you can add OCL constraints to answer Q3, but there is no straight-forward way to specify answers to Q1 and Q2.

As observed in the previous section (see **Table 7**), the answer to Q1 is usually 'no' (89%), and the answer to Q2 is split between 'yes' (62%) and 'no' (38%).

The answer to Q3 is *none* half the time (55%) – in other words most attributes have straightforward set and get behavior. The other half, there are a large number of possible characteristics to consider, since each project has unique constraints under which an attribute much conform. Two of the characteristics observe reasonably frequently are uniqueness and default values; we discuss these in Section 3.3

In the work below, we show that these answers above could be reflected in a model-oriented syntax, and in generated code. We also determined which scenarios do not make semantic or pragmatic sense; to further simplify the attribute syntax. Further discussion of code generation in Umple is in Section 5.

Is the attribute specified in the constructor (Q1)?: First, let us consider attributes that are available in the constructor (Q1.Yes). By default an attribute's value is required at construction, and the syntax to describe this scenario is to declare the attribute with no extra adornment. E.g.

```
String x;
Integer y;
```

For attributes that are not to be set during construction (Q1.No) the Umple syntax is to provide an initial value (which can be null) to the attribute, as shown below.

```
String x = "Open";
Integer y = 1;
String z = null;
String p = nextValue();
```

The initialized value follows the semantics of the target language (e.g. Java or PHP). It can either be a constant as we see for x and y, uninitialized as we see for z or an arbitrary method call (that the developer must define) as in the case of p.

Can the attribute change after construction (Q2)? By default in Umple, an attribute's value can change after construction (Q2.Yes), requiring no additional syntax to describe this scenario. A set method is generated in this case.

Attributes that cannot change after construction (Q2.No) are marked 'immutable'; the value set in the constructor cannot then be changed. No set method is generated.

```
   immutable String x;
```

As we discussed in Section 2, immutability is very useful to provide consistent semantics for equality and hashing, although not many attributes exhibited the immutable property. Part of the issue being the difficulty in specifying immutable attributes in the languages we studied (Java and C#).

There are instances where an attribute should be immutable, but it might be the case where the value is not available at the time of construction. Examples of this include application frameworks where the creation of an object is controlled by the framework and is outside the developer's control. In these cases, an initially empty object is provided to the application, to be immediately populated with the attribute data that cannot then be changed. Therefore, to support this case in Umple, we allow immutable attributes to delay instantiation by using the *lazy* keyword.

```
   lazy immutable y;
```

As in Section 3.1 the use of this syntax means that no argument is created in the constructor for the attribute. The generated code will contain a flag to track whether the object has been set yet, only allowing a single set to occur. We elaborate on immutability and the underlying executable implementation in Section 5.2.

What other characteristics does the attribute possess (Q3)? The potential characteristics are limitless. In our analysis of existing software we found three somewhat-common patterns that are incorporated into Umple.

Before we consider these patterns, we should recognize that many attributes have no explicit constraints. In general, a property like a name (or jobTitle) has no constraints apart from those enforced by the underlying language (i.e. type checking).

The first characteristic we considered is uniqueness. In databases, guaranteeing uniqueness allows for efficient searching and equality assertions; many domains also have data that is unique by design (e.g. flight numbers in an airline). In some cases, objects are automatically assigned a unique identifier upon creation, whereas in others uniqueness is checked whenever the attribute is set.

In UML, an attribute's uniqueness can be specified with a qualifier, which is really a special type of attribute. Below is an example Airline that has many RegularFlights.

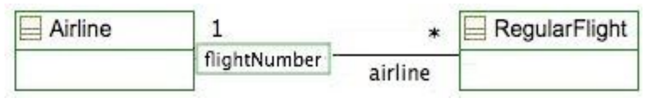

Fig. 2: Unique flightNumber on the airline association

Two RegularFlights of the same Airline should not have the same flightNumber. It is also possible to allow for global uniqueness within a system, for example an ipAddress attribute should perhaps be unique throughout the entire application.

In the cases above, the developer must define unique attributes. The example below provides a mechanism to allow the underlying system to manage the generation of valid and unique identifiers, within or outside the context of an association. The Umple syntax to describe the constraints above is shown below.

```
unique Integer flightNumber on airline;
unique String ipAddress;
```

Uniqueness for integer attributes can also be managed automatically in Umple using the *autounique* keyword.

```
  autounique Integer flightNumber on airline;
```

A *defaulted* value ensures an attribute is never *null*. Any time the internal value of the attribute is null the get method returns the default value.

```
  defaulted type = "Long";
```

Not all attributes conform to a standard set/get semantics. In addition, many member variables are not attributes, but are support variables used internally [4]. In Umple, the syntax for *internal* attributes is shown below. Internal attributes do not form part of the constructor and do not have accessor methods, allowing developers to manage this data in the way they see fit.

```
   internal Integer cachedSize = 0;
```

Finally, let us consider a *many* multiplicity. Using square bracket [] syntax, attributes can also be represented as multiple instances of the attribute type.

```
   String[] names;
```

4 Generating Code for Attributes: Existing Tools

After investigating how attributes are used in practice, we studied the code generation patterns of existing tools to see how they implement attributes. The UML modeling tools considered were identified from a Gartner report [5] and an online list of UML tools [6]. We selected four open source projects and one closed source application to analyze. ArgoUML and StarUML are two of the most active open source UML modeling tools and RSA has the largest market share; using popular tools helps to ensure that our study is relevant [5, 7].

Table 10. UML code generation tools.

Tool	Version	Source
ArgoUML	0.26.2	argouml.tigris.org
StarUML	5.0.2.1570	staruml.sourceforge.net
BOUML	4.11	bouml.free.fr
Green	3.1.0	green.sourceforge.net
RSA	7.5	ibm.com/software/awdtools/architect/swarchitect

Table 11 lists tools not considered either because they did not provide code generation (at all, or of class diagrams), or did not run in our environment.

For the tools listed in Table 10, we used a Student class with two attributes, an integer representing an id, and a list of names (represented as simple strings).

```
class Student {
  Integer id;
  String[] names
}
```

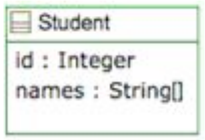

Figure 3: Student class with a simple id attribute and a list attribute

Table 11. Additional UML tools not considered for our case study.

Tool	Version	Source
Acceleo	2.5.1	acceleo.org
Jink UML	0.745	code.google.com/p/jink-uml
Hugo	0.51	pst.ifi.lmu.de/projekte/hugo
Umbrello	2.0	uml.sourceforge.net
Umlet	9.1	umlet.com
Fujaba	5.0.1	wwwcs.upb.de/cs/fujaba/
Modelio	1.0.0	modeliosoft.com
Topcased	1.2.0	topcased.org
NetBeans UML Modeling	6.7	netbeans.org
Papyrus	1.11.0	papyrusuml.org

ArgoUML: An open source modeling platform that provides code generation for Java, C++, C#, PHP4 and PHP5. Below is the code generated from Figure 3.

```
import java.util.Vector;
public class Student {
  public Integer id;
  public Vector names; }
```

The generated code provides public access to set and get the attributes; a pattern not often used in practice as it breaks encapsulation by fully exposing the internals of the class. The attributes are made directly available without the inclusion of accessor methods.

StarUML: This is an open source modeling tool. Its generated code is almost identical to that of ArgoUML, except that the import clause was missing, and names is declared erroneously as a String. StarUML's generated code hence does not account for the many multiplicity, resulting in unusable generated code.

Bouml: This is another free tool based on UML 2 that provides source code generation for C++, Java, Idl, PHP and Python. Its code is very similar to the two systems above, with the primary difference being that the attributes are declared as private and genericity is used:

```
class Student {
  private int id;
  private List<String> names;
}
```

The Bouml source code does not provide any mechanism to set or get the attributes; this code must be written by hand after code generation.

Green UML: This is another UML editor that can generate source code from a class diagram.

```
import java.util.List;
public class Student {
  int id;
  List<String> courses;
  public Student() {} }
```

Although Green UML does attempt to provide some additional source code generation to manage the various types of association multiplicities available; the results provide little if any added benefit in representing the model's intentions.

Rational Software Architect (RSA) and RSA Real-Time: IBM's Rational Software Architect is a full-fledged development environment that supports model-driven development including source code generation from UML diagrams. Its code for **Figure 3** is as follows:

```
import java.util.Set;
public class Student {
  public Integer id;
  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }
  private Set<String> courses;
  public Set<Student> getCourses() { return courses; }
  public void setCourses(Set<Student> courses)
   { this. courses = courses; }
}
```

RSA's model transformation into Java provided flexibility regarding the template patterns including (a) which Java collection to use, and (b) whether or not to include get/set methods. RSA Real Time generated the same code as RSA for class diagrams.

After reviewing several code generators it became apparent that an opportunity exists to provide better support for aligning the modeling's world of access and multiplicity constraints directly in the generated code, alleviating the modeller/developer from code duplication to ensure consistency between the model and the code. As an Umple model *is* the Umple code, such consistency is easily achieved with adequate code-generation capabilities.

5 Generating Code for Attributes using Umple

As seen earlier, existing code generating tools do not consider the many complicated facets of implementing attributes in a target language like Java. In this section, we demonstrate code generation patterns based on Umple attributes and generated into Java code.

The following example shows how one would declare attributes in the first steps of modeling a system using Umple. For conciseness, we have omitted the code comments and some additional methods not related to the attributes in the generated Java.

5.1 Basic Attributes

At the core of an Umple attribute is a name. The implications on code generation include a parameter in the constructor, a default type of String and a simple set and get method to manage access to the attribute. The attribute code in Umple is shown below, and code generated in Java follows.

```
class Student { name; }
```

```
public class Student {
  private String name;
  public Student(String aName) { name = aName; }
  public boolean setName(String aName) {
    name = aName;
    return true; }
  public String getName() { return name; } }
```

The syntax is similar to RSA generated code, and to the *simple* cases observed in the open source projects. As seen Section 2, few attributes are set in the constructor. In

Umple, this can be achieved by specifying an initial value as shown below. The generated code in Java would only differ in the constructor, and follows.

```
   class Student { name = "Unknown"; }
```

```
   public Student() { name = "Unknown"; }
```

Please note the initial value can be null, or some user defined function written in the underlying target language (i.e. Java).

5.2 Immutable Attributes

If a Student variable was declared immutable, as presented in Section 3, the resulting Java code would be the same as the basic attribute implementation, except that there would be no setName method.

By default, immutable attributes must be specified on the constructor, and no setter method is provided. But, Umple also supports lazy instantiation of immutable objects as shown below and discussed in Section 3.

```
   class Student { lazy immutable name;}
```

By declaring a lazy immutable attribute we follow the same convention whereby the *name* attribute will not appear in the constructor; but we also provide a set method that can only be called once.

```
public class Student {
  private String name;
  private boolean canNameBeSet;
  public Student() { canNameBeSet = true; }
  public boolean setName(String aName) {
    if (!canNameBeSet) { return false; }
    canNameBeSet = false;
    name = aName;
    return true; }
   public String getName() { return name; } }
```

The implementation above includes an additional check *canNameBeSet* to ensure that the variable is only set once, but should be used with caution in threaded access to avoid issues from parallel processing conflicts.

5.3 Defaulted Attributes

As discussed in Section 3, a defaulted attribute provides an object with a default configuration that can be overwritten. The code generated for Java follows.

```
   class Student { defaulted name = "Unknown"; }
```

```
public class Student {
  private String name;
  public Student(String aName) { name = aName; }
  public boolean setName(String aName) {
    name = aName;
    return true; }
   public String getName() {
    if (name == null) { return "Unknown"; }
    return name; } }
```

Below are the subtle differences between initialized and defaulted attributes. First, a defaulted attribute is specified in the constructor, an initialized attribute is not. Second, a defaulted value is guaranteed tp be non-*null*, an initialized attribute only guarantees an attribute in set to particular value in the constructor (and can change afterwards).

5.4 Unique Attributes

The Umple language currently only supports code generation for autounique attributes as shown below. The code generated for Java follows.

```
   class Student { autounique id;}
```

```
public class Student {
  private static int nextId = 1;
  private int id;
  public Student() { id = nextId++; }
   public int getId() { return id; } }
```

The implementation is constrained to non-distributed systems; but allows for a simple mechanism to uniquely identify an object.

5.5 Constant Class Attributes

A constant class level attribute is identified using the convention of ALL_CAPS. The UML modeling standard is to underline; a convention that is difficult to achieve in a development environment as most developer code is written in plain text. The code generated for Java follows.

```
class Student { Integer MAX_PER_GROUP = 10; }
```

```
public class Student { public static final int MAX_PER_GROUP = 10; }
```

5.6 Injecting Constraints using Before/After

To support vast array of other types of custom implementations of set and get methods, as well as provide a generic mechanism for managing pre and post-conditions of an attribute, we introduce the before and after keywords available in the Umple language. Let us begin with a simple example.

```
class Operation {
  name;
  before getName {
    if (name == null) { /* long calculation and store value */ }
  }
  after getName {
    if (name == null) { throw new RuntimeException("Error"); }
  }
}
```

In the code above, we are caching the derivation of the complex process to determine the value of *name*. The code is also verifying that the *getName* method always returns a value (never null). The code provided in the *before* block will be run prior to desired operation (i.e. getName) and the code block provided in the *after* block runs after (or just before returning) from the desired operation. The code generated for Java for the *getName* method is shown below.

```
public String getName()
{
  if (name == null) { /* long calculation and store value */ }
  String aName = name;
  if (name == null) { throw new RuntimeException("Error"); }
  return aName;
}
```

The before and after mechanisms can be used with any Attribute A summary of the operations is described below.

Before and after can be applied to associations, and constructors as well. This mechanism can, for example, provide additional constraints to a class, or to initialize several internal variables.

Table 12: Applying before and after operations to Attributes

Operation	**Applies To (UB = Upper Bound)**
setX	Attributes (UB <= 1)
getX	Attributes
addX	List Attributes (UB > 1)
removeX	List Attributes (UB > 1)
getXs	List Attributes (UB > 1)
numberOfXs	List Attributes (UB > 1)
indexOfX	List Attributes (UB > 1)

An operation can have several before and after invocations. This chaining effect allows each statement to focus on a particular aspect of the system such as a precondition check of inputs, or a post-condition verification of the state of the system.

It should be noted that the syntax of Umple's before and after mechanism is purposely generic with a relatively fine-grained level of control. The intent of this mechanism is to act as a building block to include additional constraint-like syntaxes for common conditions such as non-nullable, boundary constraints and access restrictions. By including before and after code injections at the model level, additional code injection facilities are possible at the model level, without having to modify the underlying code generators. For example, the immutable property discussed is implemented internally (i.e. Umple is built using Umple) using before conditions on the set methods.

6 Related Work

There is literature on code generation from UML [8-11]. In [8], an abstract class is generated for the set and get methods and an instantiable class implements operations. This adds a layer of complexity. Umple provides a more direct approach, and the generated code more closely resembles that which would be written by hand. Whereas the approach above seems guided more by the limitations of using UML.

Jifeng, Liu, and Qin [12] present an object-oriented language that supports a number of features like subtypes, visibility, inheritance, and dynamic binding. Their textual object-oriented language is an extension of standard predicate logic [13]. The approach to Umple was not to create a *new language*, but rather to enhance existing ones with a more model-oriented syntax and behaviour.

Reverse engineering tools tend to generate a UML attribute when they encounter a member variable, a practice widely adopted by software modeling tools. Sutton and Maletic [14] advocate that attributes reflect a facet of the class interface that can be read or written rather than representing the implementation details of a member variable. They present their findings on the number of class entities, attributes and relationships that were recovered using several reverse engineering tools, revealing the inconsistencies in the reverse engineering approaches. They present their prototype tool, *pilfer*, that creates UML models that reflect the abstract design rather than recreating the structure of the program.

Gueheneuc [15] has analyzed existing technology and tools in reverse engineering of Java programs, and highlights their inability to abstract relationships that must be inferred from both the static and dynamic models of the Java programs. They developed PADL (Pattern and Abstract-level Description Language) to describe programs in class diagrams. However, their proposed approach requires the availability and analysis of both static and dynamic models to build the class diagrams. In another study [16], two commercial reverse engineering tools (Together and Rose) are compared to research prototypes (Fujaba and Idea); they note that different tools resulted in significantly different elements recovered from the source code.

Lange and Chaudron [17] conducted an empirical analysis of three software systems and identified violations to a number of well-formedness rules. In one of the systems, 67% of attributes were declared as public without using setters and getters.

Experimentation with Umple [18] users reveals evidence that software developer comprehension of the code is enhanced when compared to traditional object oriented code [19-21]. Umple was deployed and evaluated in open source projects [22]

In most of the cases above, automated analysis done by reverse engineering tools resulted in vastly different perceptions about the systems being studied. Our approach, although subjective at times and error prone due to several manual steps throughout the process, attempts to provide a structured approach to reviewing, categorizing and understanding how attributes are used in practice.

7 Threats to Validity

Our empirical investigation of existing implementation of attributes has two main threats of validity; Firstly, to what extent are the seven selected projects representative of typical uses of attributes; and secondly, to what extent are attribute patterns affected by the capabilities provided by the existing programming languages.

To mitigate the risks of non-representation we were diligent to select projects in a random fashion from a large group of projects written in different languages (yet languages that we were experienced in). The process to select projects was well documented and can easily be repeated for future studies into this subject.

The second threat is to what extent the capabilities of the underlying programming language affects the types of patterns that can be observed. This threat is somewhat of an extension to our first threat, and is more difficult to mitigate, as we cannot understand what we do not know. One way to better deal with this would be to repeat the study using different programming languages with different attribute semantics.

8 Conclusion

This paper analyzed the syntax, semantics and pragmatics of attributes. We studied how attributes are used in practice; and discovered the difficulty in extracting modeling abstractions from analyzing source code. Our approach used manual inspection, which, although subject to human error is probably comparable to analysis by automated tools since there are so many special cases to be considered.

We demonstrated how attributes are represented in the Umple model-oriented language and showed the code-generation patterns used to translate Umple into Java. When compared to the code generated for attributes by existing tools, we believe our patterns have a lot to offer.

References

1. Forward, A. and Lethbridge, T. C. "Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals," MiSE, ‚International Workshop on Models in Software Engineering, 2008, pp. 27-32.
2. Forward, A., Lethbridge, T. C. and Brestovansky, D. "Improving program comprehension by enhancing program constructs: An analysis of the umple language," International Conference on Program Comprehension (ICPC) 2009, pp. 311-312.

3. "Umple language online." accessed 2013, www.try.umple.org
4. A. Sutton and J. I. Maletic. "Recovering UML class models from C++: A detailed explanation". 2007. Inf. and SW Tech vol 49, pp. 212-229.
5. D. Norton, "Open-Source Modeling Tools Maturing, but Need Time to Reach Full Potential," Gartner, Inc., Tech. Rep. G00146580, 20 April 2007, 2007.
6. "Wikipedia Listing of UML modeling tools." accessed 2013, http://en.wikipedia.org/wiki/List_of_UML_tools
7. Michael J. Blechar, "Magic Quadrant for OOA&D Tools, 2H06 to 1H07," Gartner Inc., Tech. Rep. G00140111, 30 May 2006, 2006.
8. W. Harrison , C. Barton and M. Raghavachari. "Mapping UML designs to Java". 2000. ACM SIGPLAN Notices vol 35, pp. 178-187.
9. Long, Q., Liu, Z., Li, X. and Jifeng, H. "Consistent code generation from uml models," in Australian Software Engineering Conference, 2005. pp. 23-30.
10. L. B. Brisolara, M. F. S. Oliveira, R. Redin , L. C. Lamb, L. Carro and F. Wagner. "Using UML as front-end for heterogeneous software code generation strategies". 2008. Design, Automation and Test in Europe, 2008, pp. 504-509.
11. Xi, C., JianHua, L., ZuCheng, Z. and YaoHui, S. "Modeling SystemC design in UML and automatic code generation," Conference on Asia South Pacific Design Automation, 2005, pp. 932-935.
12. H. Jifeng, Z. Liu, X. Li and S. Qin. "A relational model for object-oriented designs". 2004. Lecture notes in computer science pp. 415-436.
13. "Unifying Theories of Programming". Prentice Hall, 1998,
14. A. Sutton and J. I. Maletic. "Recovering UML class models from C++: A detailed explanation". 2007. Inf. and SW Tech vol 49, pp. 212-229.
15. Gueheneuc, Y. "A reverse engineering tool for precise class diagrams,". CASCON, 2004, ACM and IBM, pp. 28-41.
16. Kollman, R., Selonen, P., Stroulia, E., Systa, T. and Zundorf, A. "A study on the current state of the art in tool-supported UML-based static reverse engineering," Ninth Working Conference on Reverse Engineering (WCRE'02), 2002, pp. 22-30.
17. Lange, C. F. J. and Chaudron, M. R. V. "An empirical assessment of completeness in UML designs," EASE 2004, 2004, pp. 111-121.
18. Badreddin, O. "Umple: a model-oriented programming language." Software Engineering, 2010 ACM/IEEE 32nd International Conference on. Vol. 2. IEEE, 2010.
19. Badreddin, O. “Empirical Evaluation of Research Prototypes at Variable Stages of Maturity”. ICSE Workshop on User Evaluation for Software Engineering Researchers (USER), 2013 (to appear).
20. Badreddin, O, and Lethbridge T.C. "Combining experiments and grounded theory to evaluate a research prototype: Lessons from the umple model-oriented programming technology." User Evaluation for Software Engineering Researchers (USER), 2012. IEEE, 2012.
21. Badreddin, O, Forward, A, and Lethbridge T.C. "Model oriented programming: an empirical study of comprehension." CASCON, ASM and IBM, 2012.
22. Badreddin, O, Lethbridge, T. C and Elassar, M. “Modeling Practices in Open Source Software”. OSS 2013, 9th International Conference on Open Source Systems (to appear).