The Popularity Cycle of Graphical Tools, UML, and Libraries of Associations

Martin Soukup
Nortel
3500 Carling Ave.
Ottawa, ON K2H 8E9, Canada
(613)-765-6435
msoukup@nortel.com

Jiri Soukup
Code Farms Inc.
P.O.Box 344, 7214 Jock Trail
Richmond, ON K0A 2Z0, Canada
(613)-838-3622
jiri@codefarms.com

Abstract
A historical cycle has been observed where the use of graphical tools becomes critical to software development but these tools eventually fall from use as the underlying cause of complexity is removed through a new programming paradigm. This workshop attempts to take a unified view of UML-related ideas which span from high level software design (UML and MDD) to technical details of implementing reusable associations. It also identifies gaps in the existing programming languages addressed by UML class diagrams. The discussion is not language specific and applies to both C++ and Java. For additional details see www.codefarms.com/OOPSLA07/workshop

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – D*ata types and structures Control structures, Patterns.* D.1.5 [**Programming Techniques**] Object-oriented Languages. D.2.2 [**Software Engineering**] Design Tools and Techniques – Object-oriented design methods, software libraries. D.2.3 [**Software Engineering**] Coding Tools and Techniques – Object-oriented Programming. D.2.11 [**Software Engineering**] Software Architectures – Data Abstractions, Languages, Patterns. D.2.13 [**Software Engineering**] Reusable Software – Reusable libraries, Reuse models. D.3.2 [UML].

General Terms Management, Design, Reliability, Standardization, Languages.

Keywords UML, Unified Modeling Language, model-driven, model, MDA, MDE, MDD, association, reusable, class library, container, agile programming, design patterns, intrusive data structures.

1. Introduction

Figure 1 shows how these seemingly different ideas relate to each other. Heavy-framed boxes are basic observations, the box with a dashed frame connects the high level issues (top part) with the low level, implementation issues (bottom part).

OOPSLA'07 October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

Box 1: Cycles of textual programming and graphical tools

The following historical cycle has been observed: Programmers create software which keeps growing both in complexity and size until it is hard to manage. At that point programmers reach for various graphical tools to keep the complexity under control. It does not take long though and a new paradigm eliminates the source of complexity, graphical tools are abandoned, programmers return to textual programming, and a new cycle begins.

[1] describes three such cycles: Flow charts which were eliminated by the introduction of structural programming, table diagrams which were eliminated by the introduction of structures and pointers, and once popular pointer diagrams are not needed since the introduction of class libraries (STL and Java Collections).

Box 2. UML class diagrams are popular today

There is no question about it – any properly designed software project today uses UML, at least UML class diagrams. If you work in a MDD environment, UML drives the entire design and at least the first skeleton of the code is generated automatically.

Box 3. Why are UML class diagrams so popular?

It is important to understand the underlying reasons:

(a) In the current style of the object-oriented programming, classes are highly visible but their relations (associations, data structures, design patterns) are buried inside class definitions and not easy to find. UML gives us one unified view where classes and their relations have the same importance.

(b) When you program with existing libraries you think in collections (containers) and pointers (references). UML forces you to think in higher level concepts – the associations. Collections are only a subset of associations: associations include bi-directional relations among 2 or more classes, while collections (uni-directional relations between just 2 classes) are only a special case of associations.

Box 4. What will be the next paradigm?

If the observation about the popularity cycle of graphical tools is correct, the widespread use of UML today indicates that a new programming paradigm is imminent. We don't know what this paradigm is going to be, but should this paradigm supersede UML class diagrams, it would allow us to program with associations, in a style which would resemble how we program with collections today. This new paradigm would also make relations among classes (associations) first class objects, in other words these relations should be clearly visible without wading through the code.

Box 5. We need libraries of associations.

Very likely, the new paradigm will employ libraries of generic associations, but the existing OO languages, in particular C++ templates and Java generics, do not support such libraries. Generic associations must not only depend on parametric types (just like reusable collections do), but they also require additional data (pointers/references, collections, arrays) to be inserted into the classes that participate in those associations. Intrinsically, associations are intrusive data structures – an important category of data structures which have been neglected in recent years due to the widespread use of collections. One of the attempts to provide this missing feature is Aspect Oriented Programming (AOP).

When using the existing features of the commonly used OO languages, the insertion of the additional data can be arranged either through multiple inheritance or using a code generator. Since Java and C# do not have multiple inheritance, the latter method is unlikely to become popular. An attempt to do this with AOP was not successful [2].

Box 6. Structural design patterns combine associations with inheritance

This was explained in [3], and if we plan to build libraries of associations, we may right away build them in a way which includes structural design patterns.

Box 7. Expanding existing collection libraries to include associations

As a feasibility study, the organizers designed an open source library of associations based on a simple code generator which works both in C++ and in Java called IN_CODE modeling. Its main three parts can be downloaded independently from the workshop website:

(a) The C++ library plus its code generator written in C++,
(b) the Java library plus its code generator written in Java,
(c) the Layout program which generates high quality UML class diagrams from textual declarations of associations.

Each part includes documentation and a suite of test programs. The software has been tested under Windows XP, but it is coded in a general way which should run under Unix or Linux. The Layout program applies advanced VLSI CAD algorithms to make the UML diagram logical, easy to read and esthetically pleasing. Note that the code generators are coded using themselves and their own libraries – a good example of how this software can be used.

Box 8. Support of reusable associations in existing OO languages.

The important question is what features we would have to add to existing OO languages in order to support reusable associations. This is one of the issues discussed in [2]. It seems that the following features will be required:

(a) In addition to types or integers, templates (or generics) should allow parametrization of member names, for example:

```
template<class A, name abc> class MyClass {
    A abc;
    ...
};
```

(b) We will need new keywords such as *association* and *participants*. Keyword *association* declares an association, retrieves its pieces from the library, and inserts them transparently into the association classes:

```
    class Student { … };
    class Course { … };
    class Takes {
        int mark;
        int attendance;
    };
association
   ManyToMany<Student,Takes,Course> studentCourse;
```

Keyword *participants* would describes the role of classes which belong to the same association inside the library:

```
 template<class Source, class Target, class Link>
    class ManyToMany_Source {...}; //library class

 template<class Source, class Target, class Link>
    class ManyToMany_Link {...}; //library class

 template<class Source, class Target, class Link>
    class ManyToMany_Target {...}; //library class

 template<class Source, class Target, class Link>
```

```
class ManyToMany {
     participants(ManyToMany_Source,
          ManyToMany_Link, ManyToMany_Target);
     ...
 };
```

The obvious way to implement part (b) would be to add our code generator to an existing C++ or Java compiler, but there may be a better way. We have been contemplating this, but have not done any work and are looking for people interested in this part of the project.

Box 9. Impact on MDA and existing UML tools.

Since there would be a one-to-one correspondence between associations in the UML diagram and their declarations in the code, the currently used code and UML generators would be reduced to trivial programs matching the two sets.

The declarations of associations, especially if placed together as a block of code, could be used as a textual form of the UML diagram, eliminating altogether the need for using the UML class diagram, at least on small and medium sized projects. One of the workshop organizers has been comfortably working in this mode for over a decade.

Using database terminology, you can think about this block of declarations as being a schema of the data organization. This schema controls the implementation and, at the same time, gives us compact information in a form we can easily find and read. For those who still want to see the class diagram, with every compilation, we generate the UML class diagram from this schema [4].

This is really turning the existing MDA inside out. Instead of controlling the data structure by the UML diagram, the data structure is controlled from within the code by this schema. The UML diagram is automatically derived and is only an auxiliary entity. The programmer never edits it in a graphical environment – the only way you can change it is to add, remove, or modify the declarations of the associations (or the inheritance among classes). This is not only faster and safer, but it also leads to a superior diagram.

As explained in [5], model driven design isn't really about pictures but about using models when programming. All these ideas may appear rather strange to those who are using to the existing MDD. This is indeed a change of paradigm, and we anticipate a lively discussion on this topic.

Box 10. Will we go back to textual programming?

This is the main subject of this workshop, and the answer will depend on our discussion of boxes 4, 8, 9, and 13.

Box 11. Does this apply to distributed computing?

This is another important area which we would like to explore.

Box 12. UML isn't just class diagrams.

Beside the most commonly used class diagram, UML provides many other views of the design in a total of nine types of diagrams. [5]. Even if the new paradigm eliminates the need for class diagrams, the management of other views could still be a sufficient reason for using the UML in its present style. Or is it?

Box 13. Can you imagine textual representations for other diagrams?

We should discuss, specifically for each of the UML diagrams and other graphical tools in common use today

(1) What gaps in existing programming languages which create the need for these tools?

(2) What would be the impact on these diagrams of associations becoming first class entities and being available in a reusable form?

References

[1] Soukup, M. and Soukup J. The Inevitable Cycle – Graphical Tools and Programming Paradigms. In IEEE Computer August 2007, preliminary draft available at www.codefarms.com/OOPSLA07/workshop.

[2] Soukup, J. and Soukup, M. Reusable Associations. Until it is published in Dr. Dobbs Journal, the preliminary draft is available at www.codefarms.com/OOPSLA07/workshop.

[3] Soukup, J. Implementing Patterns. In the first conference on Pattern Languages of Program Design, proceedings edited by J.O. Coplien and D.C .Schmidt, Addison-Wesley 1995, pp.395-415.

[4] IN_CODE modeling – C++ and Java libraries free to download from www.codefarms.com/OOPSLA07/workshop including the documentation.

[5] Soukup, M. and Soukup, J. Graphical Tools and Language Evolution. Submitted to ATEM 2007 (4th International Workshop on Software Language Engineering), also available at www.codefarms.com/OOPSLA07/workshop.

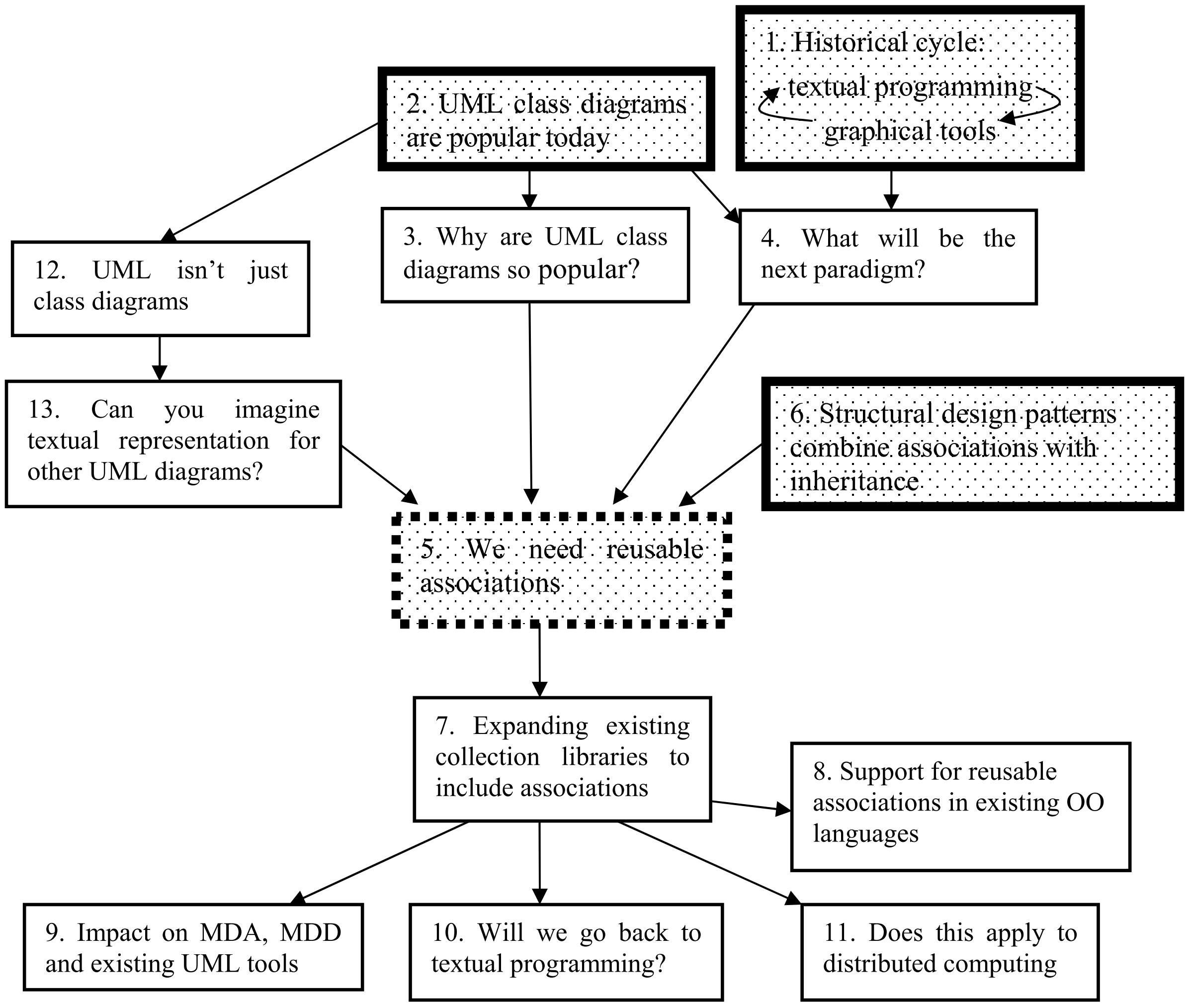
1. Historical cycle:
textual programming
graphical tools
2. UML class diagrams are popular today
3. Why are UML class diagrams so popular?
4. What will be the next paradigm?
5. We need reusable associations
6. Structural design patterns combine associations with inheritance
7. Expanding existing collection libraries to include associations
8. Support for reusable associations in existing OO languages
9. Impact on MDA, MDD and existing UML tools
10. Will we go back to textual programming?
11. Does this apply to distributed computing
12. UML isn't just class diagrams
13. Can you imagine textual representation for other UML diagrams?

Figure 1: Ideas discussed in this workshop and their mutual relations. Heavy-framed boxes are basic observations, the dashed box in the center connects the high level issues (top part) with the low level, implementation issues (bottom part).