

Graphical Tools and Language Evolution

Martin Soukup and Jiri Soukup

¹ Nortel, 3500 Carling Avenue, Ottawa, On, Canada, K2H 8E9
msoukup@nortel.com

² Code Farms, 7214 Jock Trail, Richmond, On, Canada, K0A 2Z0
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 resolved through a new programming paradigm. This paper identifies the gaps in today's programming languages addressed by UML class diagrams, a critical part of software development today. It then proposes language extensions to resolve these gaps and reviews all nine UML diagram types for other areas of possible programming language or paradigm changes.

Keywords: UML, Unified Modeling Language, model-driven, model, MDA, MDD, MDE, associations, programming language

1 Introduction

The following 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 or invent various graphical tools to keep the complexity under control. It does not take long though before a new programming paradigm is invented which eliminates the source of complexity. The graphical tools are abandoned, programmers return to textual programming, and a new cycle begins [1].

The most popular graphical tool used in software engineering today is the UML class diagram whose primary advantage over the capabilities of existing programming languages is the treatment of associations as first class entities similar to objects themselves. We have identified a simple mechanism to extend existing object-oriented languages to support associations in a manner that makes them clear and maintainable without the need for graphical tools. This also removes the common problem of desynchronization between the graphical diagrams and the actual code.

There are many other graphical tools in use today, from the large body of tools in the UML toolkit beyond the class diagram to SDL, BPEL and Model-Driven approaches that tie these tools together with domain specific languages to eliminate the need to write code manually in some applications. This paper asks what gaps in existing programming paradigms these graphical tools indicate and whether further research into the cycles of graphical tools may result in significant changes to our existing programming paradigms.

2 Cycles in the use of graphical tools

In [1], we describe three historical cycles where graphical tools became a critical part of the software development process until the underlying complexity in the programming paradigm was resolved, see Figure 1.

1. Flow charts were used to manage spaghetti logic in the code but were eliminated by structured programming.
2. Diagrams of table indices helped to manage Fortran programs but were eliminated by the introduction of C structures and pointers.
3. Pointer diagrams representing C data structures were eliminated by the introduction of class libraries.

Once these changes in programming paradigm became prevalent the use of the respective graphical tools decreased until the tools are virtually unknown.

This list focuses only on the most successful paradigms. Many others were not accepted by the programming community, due to both personal preferences and commercial interests. The attempt to move to a graphical language is as old as programming itself, rising again with each cycle. Some graphical tools, however, are intended not to address complexities in a programming paradigm but to create a new paradigm for a different set of users. We'll discuss this later.

If the observation about the popularity cycle of graphical tools is correct, the widespread use of graphical tools in software engineering, in particular the Unified Modeling Language [2], today indicates that a new programming paradigm is imminent.

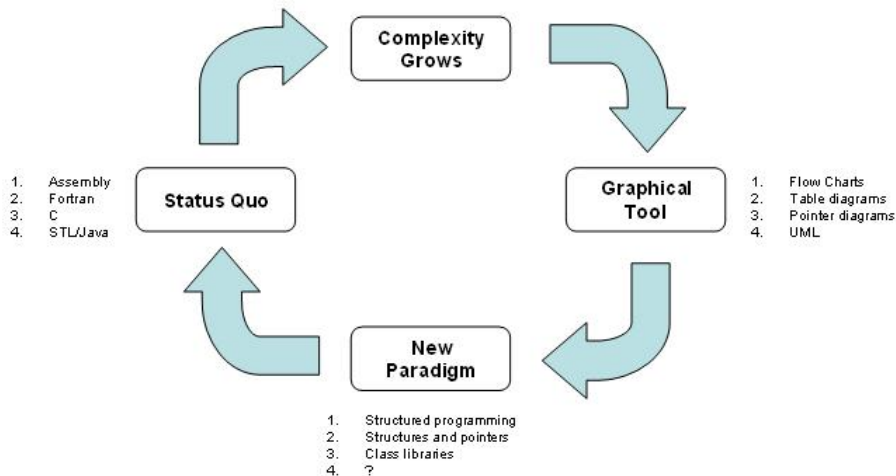


Figure 1 - Cycles in the use of graphical tools in software engineering

3 UML Class Diagrams

There is no question about it - any properly designed software project today uses UML, at least UML class diagrams. If you work in a Model Driven Engineering (MDE) environment, UML drives the entire design, usually with some additions through Domain Specific Modeling Languages (DSML) which extend the UML graphical syntax [3].

In fact, the UML class diagram is so popular that when you say “UML diagram” people assume you mean the class diagram. A UML class diagram classifies the actors in your system into a set of interrelated *classes*. Each class in the diagram may be capable of providing certain functionalities, defined by its *methods*, and may have data, defined by its *attributes* that uniquely define the class. UML class diagrams are so popular today because they provide a network representation of the associations between classes which is difficult to understand without a picture and because associations used in the UML diagram are more powerful than collections in existing languages [4]. Fundamentally, the power of the UML class diagram is that it provides a mechanism for modeling associations whereas existing programming languages only provide collections which are much less expressive.

In the current style of 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. When you program with existing languages you think in collections (containers) and pointers (references). UML forces you to think in higher level concepts - the 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.

Software complexity is a serious problem today and the prevalent use of graphical tools including UML fits this observation [3]. If this popularity cycle really exists, it would imply that the arrival of a new programming technique will eliminate or significantly reduce the use of UML class diagrams. The new paradigm would have to include the major improvements that UML class diagrams give us:

- (a) Instead of programming with collections, UML uses more general associations.
- (b) In UML, associations and classes are both treated as first-class entities.

If we could expand existing libraries such as STL or Java Collections to include associations it would improve the existing software design methodology in several ways:

- (a) It would force us to think in more general terms – in associations instead of collections and individual references/pointers.
- (b) MDE code generators would be much simpler since there would be a one-to-one match between the associations in the diagram and those in the code.
- (c) For the same reason, UML diagram generators would be easy to code and always safe to use. [4] explains the issues with generating UML class diagrams correctly from code which contains only collections and pointers/references.

[4] suggests the relatively simple addition of the association to existing object-oriented languages (C++, C#, and Java) via a keyword such as *association*. For instance:

```
association Aggregate<ParentClass,ChildClass> instanceName;
```

In addition, if we build libraries of associations we may build them in a way which would directly support structural design patterns [5]. From [6]:

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural 'class' patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together...

Rather than composing interfaces or implementations, structural 'object' patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition...

[4] discusses the implementation of an interim solution using code and UML generators and discusses two similar libraries which have been used commercially for over 10 years with excellent results.

As a feasibility study, we designed a library of associations based on a simple code generator which works both in C++ and in Java. This software, called IN_CODE is open source. Its three main parts can be downloaded independently: The C++ library including its code generator written in C++, the Java library including its code generator written in Java, and the Layout program which generates high quality UML class diagrams from a simple textual definition of associations [5].

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 with minimal to no porting under UNIX or Linux. The Layout program applies advanced VLSI CAD algorithms to make the UML diagram logical, easy to read and aesthetically pleasing. The code generators are coded using themselves – a good example of how these libraries can be used.

There are open questions as to whether we need to extend the current concept of associations to support performance or caching behaviour, referential integrity, associations between distributed objects (e.g. 3GPP IP Multimedia Subsystem [7]), to take into account persistency models (e.g. Java Persistence API [8]), etc. This is an open research question to our knowledge.

4 A broader view of UML

UML, of course, is much more comprehensive than just the class diagram (the static design view). We have not conducted detailed analysis of the key advantages that the other UML diagrams offer and what new paradigms might be required to close those gaps in our existing paradigms and languages. We present a high-level

description of each UML diagram type here with some thoughts on what activities and trends might be related to addressing those gaps, where we know of them. There are surely many others and we would be interested to hear about them. Refer to Figure 2 for the UML classification model for the 9 diagram types.

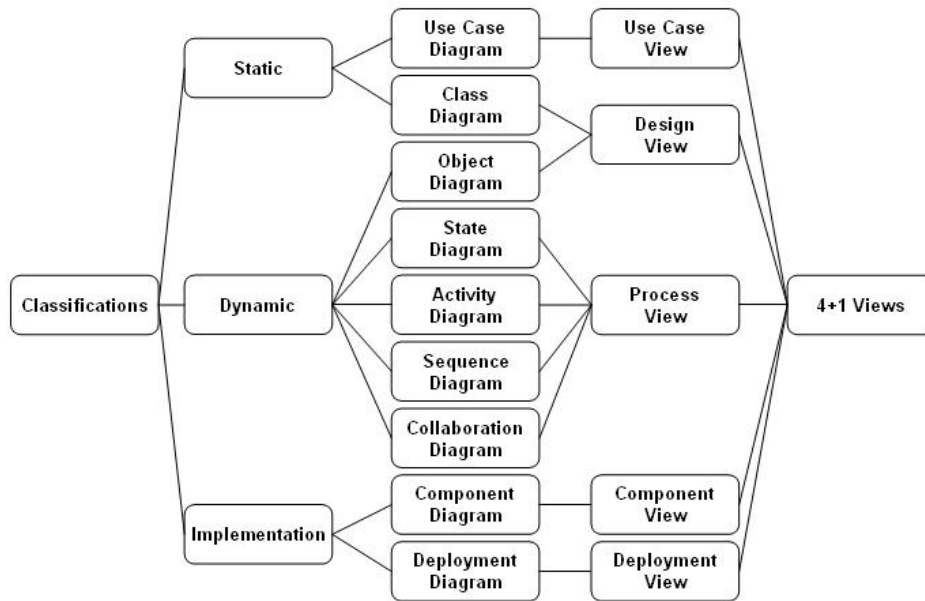


Figure 2 - Classification of UML Diagram Types

Use case diagrams identify the actors in the system and the processes that are supported by them, the use cases. This is really the definition of what the system processes are and the high-level description of the system function. Business Process Execution Language (BPEL) and graphical tools built around BPEL allow business processes to be defined from an existing set of component capabilities.

Class diagrams were described and discussed above. It is worthwhile noting that class diagrams are also commonly used for defining interfaces and their associations in message based and Service Oriented Architectures.

Object diagrams are class diagrams that describe object instances in a running system capturing their state and associations at a given point in time, typically key points in a use case.

State diagrams define the states that objects undergo during their lifecycle and the allowable transitions between states. The Pattern Template Library provides both generic associations and design patterns, including the Finite State Machine, and has been successfully used in numerous projects to model system and object state without the need for graphical tools to understand the model [9]. Petri Nets and Coloured Petri Nets are an alternative graphical state modeling language.

Activity diagrams capture the process flow of the system via activities, actions, transitions, initial and final states and guard conditions.

Sequence diagrams show the interaction of different components in the system in time through messages between the components (method calls when the components are objects). This kind of system understanding becomes critical in large scale systems integration activities and Service Oriented Architectures and is related to business process automation languages such as BPEL.

Collaboration diagrams group together interactions between objects and help identify all the possible interactions an object has with other objects.

Component diagrams represent the parts that make up the system. This typically defines the logical construction of components in both the design and build processes. Again, business process automation languages such as BPEL can show available components graphically and define their relationships. Maven is a key tool used widely in both open source and commercial development projects for textually describing system components to provide the logic to build systems to construct them.

Deployment diagrams capture the configuration of the runtime elements of the application in deployment. The operational costs of deployment and maintenance are likely the driver that has created so much activity in this space. Numerous standards and tools exist in the specification of deployment configuration. They are likely not yet in wide use across industries due to immaturity, competition between standards and commercial competition. Tools such as those from Altiris, IBM and others allow for drag-and-drop deployment via deployment modeling done in proprietary scripting languages. At least two standards also exist for full-scale deployment modeling; W3C's Installable Unit Deployment Descriptor and OASIS' Solution Deployment Descriptor. In addition the Java Enterprise Edition specifies a *deployment descriptor* for application deployment modeling in a container context.

When considering the use of graphical tools commonly used today in the software design process it is important not to focus solely on UML. Specification and Description Language (SDL), network deployment diagrams and many others are in common use today and may or may not identify the same programming paradigm gaps as those signified by UML's diagrams.

5 Model Driven Engineering

Model Driven Engineering strives to simplify and improve the quality and speed of the software design process through abstraction and formal modeling [10]. This is consistent with the questions asked and ideas posed in this paper. Whether we use textual models or graphical models doesn't change the purpose or the principles [11].

6 Conclusion

Will we return to textual programming? We postulate that we will continue to return to textual programming for most software engineering tasks until the next complexity wall is reached and new graphical methods are invented. A workshop on this topic will be held at OOPSLA 2007 [5].

Further research is needed into the key language issues and into reasons why individual graphical tools are used today. We should supplement rather than extend the software design process and, for that, we need creative solutions to programming language design and new paradigms to address them.

References

1. Soukup J. & Soukup M.: The Inevitable Cycle: Graphical Tools and Programming Paradigms. In: IEEE Computer, August 2007 (2007)
2. Rumbaugh J., Jacobson I., Booch G.: The Unified Modeling Language Reference Manual. Addison-Wesley (1998)
3. Schmidt D.C.: Model-Driven Engineering. In: IEEE Computer, February 2006, pp.25-31 (2006)
4. Soukup J. & Soukup M.: Reusable Associations. Submitted to Dr. Dobbs July 2007
5. Soukup J. and Soukup M.: The popularity cycle of graphical tools, UML, and libraries of associations. OOPSLA 2007 Workshop, October 2007, Montreal Canada, <http://www.codefarms.com/OOPSLA07/workshop/index.htm> (2007)
6. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
7. Camarillo G. & Garcia-Martin M-A.: The 3G IP Multimedia Subsystem (IMS): Merging the Internet and Cellular Worlds, Second Edition. John Wiley & Sons Ltd. (2006)
8. Sriganesh R.P., Brose G., Silverman M.: Mastering Enterprise JavaBeans 3.0. John Wiley Publishing Inc. (2006)
9. Soukup J.: Intrusive Data Structures. In: C++ Report three parts between May and October 1998 (1998)
10. Meservy O.T. and Fenstermacher K.D.: Transforming Software Development: An MDA Road Map. In: Computer, Sept. 2005, pp.52-58 (2005)
11. Soukup M.: Model Driven Architecture: Feasibility or Fallacy. In: XML 2004 Proceedings, <http://www.idealliance.org/proceedings/xml04/abstracts/paper200.html>
12. Soukup J.: Implementing Patterns”, pp. 395-412, Pattern Languages of Program Design (edited by Coplien J.O. and Schmidt D.C.). Addison-Wesley (1995)
13. Soukup J.: Taming C++: Pattern Classes and Persistence for Large Projects. Addison-Wesley (1994)

Martin Soukup is a software design manager at Nortel. His research interests include model-driven design, security, XML, video, and distributed systems. He received an MSc in Information Technology from the University of Liverpool and is a member of the IEEE. Contact him at msoukup@nortel.com.

Jiri Soukup, cofounder of Cadence Design Systems, is president of Code Farms Inc. His research interests include problems of high complexity, model-driven architecture, automatically persistent data, libraries for associations and design patterns. He received a PhD in technical cybernetics from the Czech Technical University, Prague and is a senior member of the IEEE. Contact him at jiri@codefarms.com.