The Drawbacks of model-driven Software Evolution

by
Harry M. Sneed
ANECON GmbH, Vienna
SORING Kft, Budapest
harry.sneed@anecon.com

Abstract:
This short paper is an essay on the drawbacks of model driven software evolution which apply equally well to model driven software development. The idea of automatically generating code changes from a UML type model is equally enticing as that of automatically generating whole components from such a model. The drawback is that there is then nothing to test against, since there is only one description of the system, the model. This violates the principles of software verification and validation, according to which correctness can only be demonstrated by comparing two independent descriptions of the same solution. For this reason, the author proposes another interpretation of model driven evolution, one in which the requirements model serves as a basis for propagating changes to both the code and the test, along two independent paths. The UML type system design could then be generated from the code and not [vice] versa
Keywords: Evolution, Change Management, Modeling, Verification and Validation.

1. Software Systems

In respect to the concept of model driven software evolution, one must define what kind of software is being evolved and what kind of a model is driving the evolution. There are many kinds of software systems and also many kinds of models. [BELE1975]

The types of software systems range from real time embedded systems for driving machines to distributed information management systems serving organizations. Software systems are of very different types serving very different purposes. The rate and scope of evolution also varies tremendously, depending on the nature of the system. Embedded systems and systems driving mechanical processes are closely linked to the hardware. Their rate and scope of change is restricted to the device they are driving. Business systems are embedded within an organization. Their change rate is determined by organizational change which is both, frequent and significant.

Telecommunication systems are somewhere in between, since they are to some extend dependent on the technology and are on the other hand driven by business requirements. Thus, depending on the system type, evolution can vary to a great extend, both in scope and in frequency. How much one invests in evolution is determined by the type of system one has.

2. Software models

Models are also of a wide variety. Usually, when one thinks of a model, one thinks of a graphic representation along the lines of a construction plan with entities and connecting arrows. Earlier, models were made up of tree diagrams and flow diagrams. Later, models consisted of entity/relationship diagrams sequence diagrams and state transition diagrams. The unified modeling language attempts to unify all of the previous diagram types plus some new ones into a common, all encompassing set of diagrams intended to describe a software system. [DORI2003]

Diagrams or pictures are not the only way to describe a system. One can argue that the program code is also a description of the system, albeit a very low level one, but it contains all of the details required to really understand the system. Abstraction means suppressing details, which also means losing them. Thus, a model can only be a partial description whereas the code is the complete description. The sum of all the program sources - the workflow languages, the interface description languages, the database schemas, the header files and the classes make up a comprehensive and complete description of the system developed. [BOCK2003]

On the other hand, there are the specification languages like Z, VDM and OCL which are high level descriptions of the system to be developed. These languages are closer to the way a user would view the system provided he is familiar with set theory. With them one can describe a complete solution independent of the technical implementation. Whether the user can understand them is another question.

Finally, there is the natural language itself, which is also an abstract description of a proposed or existing system. More often, requirement specifications are formulated in some restricted form of a natural language. The history of mankind has proven that natural language is the preferred means of describing situations whether or not they physically exist or only exist in the minds of men. Since situations and minds differ so do the natural languages. Of course, there are situations, when diagrams are more expressive than words, in which case diagrams can be used to supplement the words. Good descriptions of real or imagined phenomena are more often combinations of text and diagrams, which is how most software systems are described. [SEID2003]

3. Top-Down and Bottom-Up Software Evolution

The problem with software evolution is how to keep the description of the system synchronized with the system itself, i.e. how to synchronize the code with the model, when the system is changing rapidly and significantly. The implied goal of model driven software evolution would be that the model is changed and that the changes are automatically propagated to the real code as depicted in Figure 1.

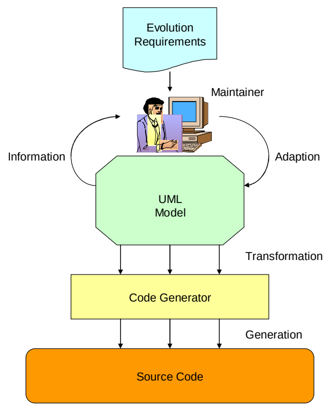

Figure 1: Top Down Model-driven Approach

This assumes some kind of automatic transformation between the higher level description of the system and the lower one. If a function is added to the model, then that same function pops up somewhere in the code or perhaps in many places in the code. The prerequisite to really applying such an automatic transformation is that the modeling language is closely related to the programming language, i.e. the higher level description of the software is not far removed from the lower one. The further the modeling language is from the code being modeled, the more difficult and error prone is the transformation. [HAR2004]

The model driven approach builds on the classical top-down approach to software development, which is in itself a fallacy. The advocates of this approach have a naïve belief in the ability of commercial developers to understand what they are doing. In reality they do not have the slightest idea. They play around with a problem until they have found an acceptable solution. As Balzer wrote "in actual practice development steps are not refinements of the original specification, but instead redefine the specification itself ... there is a much more intertwined relationship between specification and implementation than the standard rhetoric would have us believe". [BALZ1982]

This author has had the opportunity to observe commercial developers at work for almost 40 years. He finds it difficult to accept the hypothesis that developers working with UML tools understand the problems they work on any better than they did 20 years ago working with CASE tools based on structured analysis and design. The problem then was the human operator and it is still the problem. Developers, especially the run of the mill programmers working in industry, are not able to conceptualize the solution to a complex problem no matter what language they are using to express themselves or what tool they are using to implement the language. As Michael Jackson put it so blatantly "System requirements can never be stated fully in advance, not even in principle, because the user does not know them in advance - not even in principle". [JACK1982]

A contrary approach is the bottom-up one. The changes are made to the low level description of the system, i.e. to the code itself, and are then propagated by means of reverse engineering techniques to the upper level description. Thus, if an interface is added to the code, that interface description will be updated in the model automatically. This approach ensures that the model is always a true description of the system itself. However, here too, for this to work, the modeling language must be closely related to the programming language. All of the constructs in the programming language must have some equivalent in the modeling language otherwise they will be distorted, which is often the case in translating natural languages. [SEL2003] Figure 2 depicts the bottom-up approach.

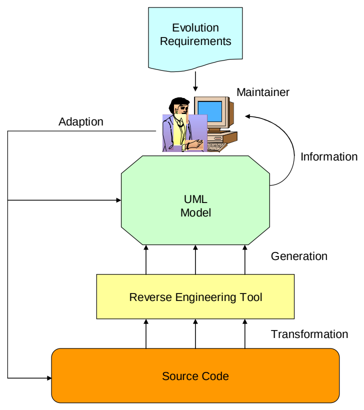

Figure 2: Bottom-Up Model-driven Approach

The biggest drawback of both, the top-down and the bottom-up approaches to model driven software evolution is that in both cases, there really is only one description of the system. The other description is only a translation of the original one in a different language. In the case of the top-down approach, the original description is the model. The code is merely a copy of the model in another form. In the case of the bottom-up approach, the code is the original description and the model is derived from it. As such, the model is only another, somewhat higher level description of the code, both of which are descriptions of the real systems. [SNED1988]

The question posed here, is what is easier to change - the graphical, higher level description, or the textual lower level description. Theoreticians would argue that it is easier and better, to change the diagrams or the higher level notations. Practicing programmers would argue that it is easier and better to change the code or the low level notations. Both have good reasons for their choice. After 15 years of research on automatic programming, Rich and Waters came to the conclusion that "to write a complete specification in a general-purpose specification language is seldom easier and often incredibly harder than writing a program. Furthermore, there has been little success in developing automatic systems that compile efficient programs from specifications". [RICH1988] As of now, this author has found no reason to believe that the program generators have become significantly better.

Theoreticians will claim that diagrams are easier to comprehend and offer a better overview. Practitioners will argue that the code is the most exact description of what is going on and that it offers a more detailed view. Besides, the practitioner will argue that he really does not know what will happen when he invokes a change, so he must first try out many variations until he finds the right one. This goes much faster in the code itself. Being a practitioner, this author tends to share the programmer's view. In most cases it is that last 10% which makes the difference. [MATH1986]

The top-down approach assumes that the system maintainers know what they are doing, that they are able to project the affects of their model changes on the underlying code. This authors knows that they don't. Maintenance programmers are by nature hackers. When they have a change to make they experiment with different variants of that change until they have found one that fits. Consequently, software evolution at the lowest level is a trial and error process, which is often repeated many times before the right solution is found. For this reason, it is open to debate which approach is really better. It may depend on the system type and the knowledge of the maintainer. [GLAS2004]

4. The need for a dual approach

However, this is not the point here. The point is that both approaches are based on a single description of the software system, since the other description is only a translation. That fact is what makes both model driven development and model driven evolution unacceptable for verification and validation. To verify a system, i.e. to prove that a system is true, one needs at least two descriptions of that system, which are independent of one another. Testing implies comparing. A system is tested by testing the actual behavior with the specified behavior. [DEMI1979] If the code is actually derived from the specification, then the code is only that same specification in another notation. The test of the system is then in fact only a test of the transformation process. To assure the quality of a system it is necessary to follow a dual approach as depicted in Figure 3.

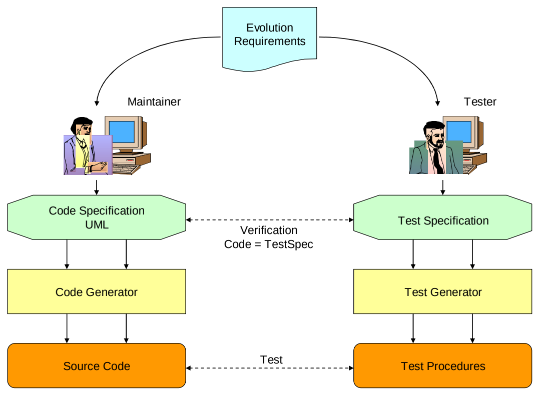

Figure 3: Dual Approach

In testing a system, the test cases and test data must be derived from another description of the system, other than that from which the code is derived. That means that there should be two independent descriptions of the final solution, preferably in two different languages. The one should be in the language of the developers, the other in the language of the users and that is their natural language. [FETZ1988]

5. Using the requirements as a model

The developers may choose between using a programming language or a modeling language or they may use both. In either case, the users will have their own languages which will be some form of natural language. For processing as well as for documenting the requirements, it will be necessary to put the natural language into some kind of regular form. This could be a combination of texts, tables and diagrams. The requirements themselves should be listed out as a series of numbered texts. This list of requirements should be enhanced by a set of tables including

- a table of user interfaces
- a table of system interfaces
- a table of business objects
- a table of business rules
- a table of business processes
- a table of system actors
- a table of system resources and
- a table of use cases. [ROB1999]

The use cases should be described with their attributes in separate tables enhanced by a use case diagram.
In all of the tables the texts should be written out as full sentences or as proper names. The same applies to the requirements. Every requirement should be a paragraph with two or more full sentences. In the use case descriptions, the steps of the main and the alternate paths should be listed out together with the pre- and post conditions, the triggers, the actors, the rules, the inputs and outputs and the exceptions. The business rules should be explicitly stated as logical conditions or arithmetic expressions. The objects should be described with their attributes. [ERIK2000]

All of this information can be contained in a single comprehensive document or it can be distributed among many separate documents. The important thing is that the documents are both readable for the user and readable for a program. The user must be able to check if the document really reflects what he wants. Since users normally understand their own natural language that means what they want has to be described in natural language. The program must be able to extract test cases from the text to check the text for consistency and adherence to rules, to derive metrics for making cost estimations and for providing the design of a requirement based system test. Programs can only work on regular grammars. Therefore the requirement specification must fulfill both prerequisites. If it can do this, the requirements specification can serve as a basis for both the test and the development. [LAMS1998]

Using the same set of requirements documents as a base line or, as the test pioneer Howden once put it, as an oracle, the developers will produce either a higher level or a lower level description of the system while the testers produce another description of the same system in the form of test cases and test data. In this way, two separate interpretations of the requirements will be made, one from the viewpoint of developers and another from the view point of the testers. At the end, these two independent interpretations will be compared against one another to determine the correctness of the system. [HOWD1987]

6. Requirements driven software evolution

So where does that leave us, as far as evolution is concerned? It leaves us with the necessity of maintaining and evolving two separate descriptions of the software being evolved - one in a normalized natural language and one in a graphic or programming language. It is absurd to believe that a graphic or modeling language description will ever be able to replace the natural language one. For that, we would need to replace our users by software technicians. This applies to other fields as well. Few house builders are able to understand the construction plans created by the architect, but they still can visualize what they want and express it with pictures and natural language.

Therefore, in software evolution, there is no real need of a modeling language like UML. If the maintenance personnel would like to have an overview of their code in graphical form, they can use a reverse engineering tool to have it on demand. For sure, the users, managers and testers will never ask for it. They will stick to their natural language description. The maintainers are usually best served by a software repository and a flexible query facility which provides them with information on demand. No study on software maintenance has ever proved that UML diagrams contribute to reducing maintenance costs. So why maintain them? [MUNS2005]

In the end, model driven software evolution will boil down to a requirements-driven software evolution. It will be absolutely essential to evolve the requirements documents and not the UML. UML is not a requirements language and never was intended to be. It is a language for software technicians and not for users. It may help to improve the communication between developers, but aside from the use cases, it does not help in promoting communication with end users. For that there is no substitute for well structured natural language. The requirement-driven model is depicted in Figure 4.

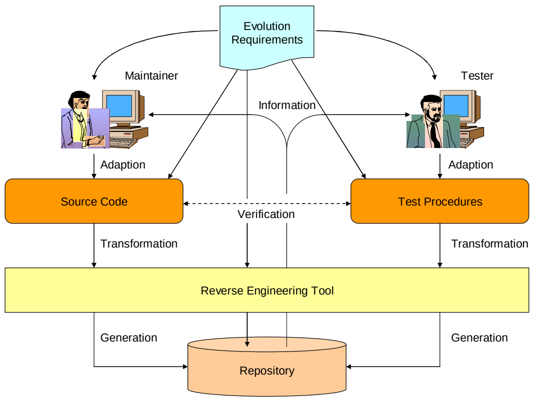

Figure 4: Requirements-driven Approach

In the requirement model three descriptions of the system exist – the requirements document, the source code and the test procedures. For every change or enhancement, both the source code and the test procedures must be adapted. It is important that this is done by two different persons with two different perspectives – the programmer and the tester. On the side of the code, the programmer will have to map the requirement changes onto the source code. On the side of the test, old test cases will have to be altered and new test cases will have to be generated. Both the programmer and the tester will require information on the impact of their changes. For this purpose, there should be an invisible model of the software system contained in a software repository. This repository should be populated via source code analysis as well as by requirement analysis and test case analysis. In this way, any requirement change can be linked directly to the code units affected.

In software maintenance and evolution the testers are the representatives of the users. Thus there are two groups working along side each other – the maintenance team and the test team. Both can get any information they want about the system on demand through a query facility. Their questions should be answered directly without forcing them to ponder through a series of UML diagrams. They will then be able to see what the impact domain of the new or changed requirement is, but nothing will be updated automatically since this would violate the principle of comparing two independent solutions.

The maintainer will have to alter or enhance the code manually, based on his interpretation of the change request. The tester will alter existing test cases or create new ones, based on his own particular interpretation of the requirement. In this way, the duality is preserved. The costs may be higher, but that is the price of quality. To have only a single description of a system which satisfies both the testers and the developers is not only an illusion, but also a gross violation of the verification principle. The goal should be to maintain two independent descriptions in two different languages. The price is that of preserving separate user/tester and developer views of an evolving system.

7. Conclusion

In light of the need to preserve software correctness, model driven software evolution should never come to mean generating code changes from UML type models. It should at best mean maintaining the user requirements model and propagating changes in it to both the source code and the testware by two different routes, either by two different tools or by two different persons, working parallel to one another. The software engineering community is called upon to stop this mania of reducing the time and cost of software maintenance and development at any price. Quality and long range stability must have precedence over quantity and short range benefits which soon turn to long term liabilities.

8. References

[BELE1975] Belady, L. /Lehman, M.: „The Evolution Dynamics of Large Programs“, IBM Systems Journal, Nr. 3, Sept. 1975, p. 11

[DORI2003] Dori, D.: “Conceptual Modelling and System Architecting”, Comm. Of ACM, Vol. 46, Nr. 10, Oct. 2003, p. 63

[BOCK2003] Bock, C.: “UML without Pictures”, IEEE Software, Sept. 2003, p. 33

[SEID2003] Seidewitz, E.: “What Models mean”, IEEE Software, Sept. 2003, p. 26

[HAR2004] Harel, D./Rumpe, B.: ”Meaningful Modelling – The Semantics of Semantics”, IEEE Computer, Oct. 2004, p. 64

[BALZ1982] Balzer, R./Swartout, V.: “On the inevitable intertwining of Specification and Implementation”, Comm. Of ACM, Vol. 25, No. 7, July, 1982, p. 27

[JACK1982] Jackson, M./McCracken, D.: “Life cycle Model considered harmful”, SE-Notes, Vol. 7, No. 1, April 1982, p. 11

[SEL2003] Selic, B.: “The Pragmatics of Model-Driven Development”, IEEE Software, Sept. 2003, p. 19

[SNED1989] Sneed,H.: “The Myth of Top-Down Development and its Consequences for Software Maintenance”, Proc. Of 5th ICSM, IEEE Computer Society Press, Miami, Nov. 1989, p.

[RICH1988] Rich, C./ Waters, R.: “The Programmer’s Apprentice”, IEEE Computer, Nov., 1988, p.15

[MATH1986] “The last 10%”, IEEE Trans. On S.E., Vol. 12, No. 6, June, 1986, p 572

[GLAS2004] Glass, R.: “Learning to distinguish a Solution from a Problem”, IEEE Software, May, 2004, p. 111

[DEMI1979] Demilo / Lipton / Perlis: “Social Processes and Proofs of Theorems and Programs”, Comm. Of ACM, Vol. 22, No. 5, May, 1979

[FETZ1988] Fetzer, J.: “Program Verification – The very Idea”, Comm. Of ACM, Vol. 31, No. 9, Sept. 1988

[ROB1999] Robertson, S./Robertson, J.: Mastering the Requirements Process, Addison-Wesley, London, 1999

[ERIK2000] Erikson, H-E./Penker, M.: Business Modeling with UML, OMG Press, John Wiley & Sons, New York, 2000

[LAMS1998] Lamsweerde, A. / Willemet, L.: “ Inferring Declarative Requirements Specifications from Operational Scenarios”, IEEE Trans. on S.E., Vol. 24, No. 12, Dec. 1998, s. 1089

[HOWD1987] Howden, W.: Functional Program Testing and Analysis, McGraw-Hill, New York, 1987

[MUNS2005] Munson, J. / Nikora, A.: „“An Approach to the measurement of Software Evolution“, Journal of Software Maintenance and Evolution, Vol. 17, No. 1, Jan. 2005, p. 65