NSE Extracts Software Models from Source Code
- Software Modeling Revolution Based on Complexity Science

Liana Ye¹, Jay Xiong²
PeaceNames.com, USA¹, NSESoftware, LLC, USA²

Abstract

Software specification and implementation are intertwined. Model driven software development is considered harmful if the models are outcomes of reductionism and superposition of linear software development processes. Successful software products are outcomes of a non-linear approach. This paper introduces a nonlinear, holistic, and dynamic software modeling approach based on complexity science, driven by platform-independent Java source code or platform-dependent programming language source code, called NSE modeling. NSE modeling enables software design automation from stubs in top-down pre-coding design, and enables the coding process to be bottom-up to further design, and effectively incorporates model-driven software design into a non-linear process.

Keywords: software modeling, MDE, MDA, MDD, software requirement engineering, software design, coding, testing, quality assurance, maintenance

1. Introduction

“A model of a system is a description or specification of that system and its environment for some certain purpose” (OMG). “A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.”[1]

Claimed by Jim Arlow and Ila Neustadt in their book, MDA, a set of large software system standards, “The future of UML may be a recent OMG initiative called Model Driven Architecture (MDA). MDA defines a vision for how software can be developed based on models. In MDA, software is produced through a series of model transformations aided by an MDA modeling tool. An abstract computer-independent model (CIM) is used as basis for a platform-independent model (PIM). The PIM is transformed into a platform-specific model (PSM) that is transformed into code.”

Critic of MDA Harry Sneed pointed out [3]: “Model driven considered harmful:

* Model-driven tools magnify the mistakes made in the problem definition;
* Model-driven tools create an additional semantic level to be maintained;
* Model-driven tools distort the image of what the program is really like;
* The model cannot be directly executed. It must first be transformed into code which may behave other than expected;
* Model driven tools complicate the maintenance process by creating redundant descriptions which have to be maintained in parallel;”

In summary, if a UML design can really replace the programming code as envisioned by some, then it becomes just another programming language. The question is, which is easier to change, the design documents or the programming language. Harry Sneed: “This depends on the nature of the problem and the people trying to solve it. If they are more comfortable with diagrams, they can use diagrams. If they are more comfortable with text, they should write text. Diagrams are not always the best means of modeling a solution. A solution can also be described in words. The important thing is that one model is enough – either the code or the diagrams. They should be reproducible from one another.”

2. ADM Approach Drives to NSM revolution

Many excellent software development approaches or methodologies have been developed to streamline a process. All these tools can be used by people for holistic examination for a software system in development. But people do forget some details. ADM group has provided KDM view from existing large systems, which are developed from many nonlinear processes in the past. ADM approach sets the direction for NSM revolution but extracts from machine code, creating another layer under existing source code. In fact, it throws away existing
source code, which is the result of successful past non-linear processes. Automating model creation from legacy source code systems, studying and improving these working models from the past, and incorporating new requirements into code generation can revolutionize software engineering.

From practical experience, the existing software engineering paradigm has always been divide and conquer, thus an outcome of linear thinking, reductionism, and superposition.

Fig. 2 MDA Agile software development process [5]

Products woven together with these approaches are un-maintainable. Any requirement change or code modification will make the products unstable as current software processes ever larger amount of dynamic data, as experience has already shown. Today, most critical software defects are introduced into a software product in the requirement development phase and then the design phase. Yet dynamic testing of the product is performed after coding. The National Institute of Standards and Technology’s study [6] said: “Briefly, experience in testing software and systems have shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.” Working models from legacy systems need to be re-examined to satisfy these new requirements.

2.1. ADM approach has problems to work out

According to ADM structure in Fig. 1, ADM has to rewrite the source code of a legacy system in KDM. If we use the code generated by MDA, it will take a large amount of resources to cover an existing legacy system in detail, unless the models are ugly enough with a lot of detailed information making these models hard to view.

Either MDA or ADM may provide two sources approach to software modeling (Fig. 3) with one in models or diagrams format for people to understand a complex software system, and the other in textual format, or source code, for computers to interpret the system. There is a big gap between the two sources.

Fig. 3 Two-Source Approach in software modeling

Traditional code-driven engineering approaches do not support software modeling for high-level abstraction, making the developed software product hard to understand and hard to maintain.

Similar to ADM legacy system approach, a proposed Nonlinear Software Modeling (NSM) [4] takes only source code as the single source for both human and machine understanding, instead of avoiding the old code, as both MDA and ADM are doing. NSM creates models of a complex software system with colorful, dynamic, virtual, interactive, traceable, linkable, auto-convertible, accurate, precise display, assuring model consistency with the source code.

2.2. Requirements to satisfy NSM

The foundation of NSM is laid in Fig 4. To count for all the elements indicated in the framework and using existing technologies and best practice today, the following requirements to satisfy NSM are feasible.

(1) Models or diagrams should be meaningful for describing both high-level abstraction and low-level program logic from the same source.
(2) Models or diagrams should be holistic, colorful, interactive, dynamic, and traceable.
(3) Programming sources should be stable and platform-independent, such as Java-DSL or C++, without changes for high-level abstraction to generate new models or diagrams.
(4) Tools should be fully automated to generate models or diagrams directly from such source codes.
(5) Models or diagrams should not take a large amount of space in static storage.
(6) Bi-directional traceability must be supported.
(7) Developed software products should be truly maintainable, counting in all related models, diagrams, documents, test cases, and the source code.

3. NSM solution

Shown in Fig. 5 in NSM, one source is used for both human understanding and computer understanding of a software product.
The models and diagrams are automatically generated from their source code, either stub modules (having an empty body, or only a set of function call statements), or a regular program, through reverse engineering. The generated models, diagrams and the source code are fully traced. Further, dynamic design and coding are fully integrated in a non-linear way as shown in Fig. 6.

4. 3J-graphics

Three types of models and diagrams are created called J-Chart, J-Diagram, and J-Flow.

4.1. J-Chart is not only used to represent class inheritance relationships, function call graphs, and the class-function coupling structure of a software product, but also is used to display the orders of incremental unit testing or related test coverage, where quality data in bar graphics is overlaid on each module-box, to show overall results of the test. J-Chart is easy for software modeling, system understanding, inspection, test planning, test result display, re-engineering, and software maintenance. J-Chart can be automatically generated from a stub program of “Bone Programming” for high-level abstraction, or a regular program including legacy programs. J-chart notations are shown in Fig. 7.

4.2. J-Diagram notations are shown in Fig. 8. J-Diagram is automatically generated from source code in all levels, including class hierarchy tree, class structure diagram, and class member function logic diagram with un-executed class/function/segment/condition outcomes being highlighted. J-Diagram is automatically linked together for an entire software product to make the diagrammed code traceable in all levels. J-Diagram can be automatically converted into J-Flow diagram. J-Diagram is particularly useful in Object-Oriented software understanding, inspections, walkthroughs, testing, and maintenance.

4.3. J-Flow The majority of traditional control flow diagrams are un-structured. They often use the same notation to represent different program logic, and cannot display logic conditions and source code locations. J-Flow diagram is Object-Oriented and structured. It uses different notations to represent different logic with capability to show logic execution conditions and corresponding source code locations. J-Flow is particularly useful in logic debugging, path analysis, test case, code correspondence analysis, and class/function level test. The test coverage
result is displayed with unexecuted elements (paths, segments, and unexecuted condition outcomes) highlighted.

J-Flow diagram in Fig. 9 can be converted to and from J-Diagram automatically. In NSM, J-Flow not only shows program control flow, but also shows the best path for testing on mostly untested branches. Its execution conditions are automatically extracted for semi-automatic test case design in unit testing.

J-Flow plays an important role in software traceability among all related documents, such as requirements specification, models, diagrams, test requirements, test cases and source code. All traceability operations use J-Flow forwardly and backwardly. When a user selects a requirement and clicks a related test case in a window, that implementation is forwarded and that window will automatically show the test case in blue, while its corresponding test coverage result is shown in J-Flow in another window with its classes, functions, and branches are tested and highlighted in red.

The dynamic and interactive 3J-graphics, Object Oriented charts, logic diagram control flow diagram, and a chart generator, are a trinity. The trinity is always running when a chart is shown. 3J-Graphs are generated directly from the source code of the platform-independent Java programs or a platform-dependent program written in C, C++, or VB. With the 3J-Graphs and the corresponding tools and languages, high-level abstraction, such as Actor and Action notations similar to Use Case diagram of UML can be generated.

5. HAETVE [hayn-tiv] Technique

HAETVE means Holistic, Actor-Action and Event-Response driven, Traceable, Visual, and Executable techniques for dynamic software modeling. With HAETVE the graphical notations for representing an actor and its action using Java language are shown in Fig. 10 A, where the notation used for representing an actor is designed for representing a recursive program module below:

```java
public class notations {
    public static void Bank_Customer ()
    {
        // Bank_Customer ()
        // }
    }
    public static void Deposit_Money ()
    {
        // }
    }
}
```

Java is a platform independent programming language. Results obtained in modeling from Java should be independent from target languages and platforms. If there is a need, the stub java source code can be transformed to a target language source code.

The notations for representing an actor and its action using C/C++ programs are shown in Fig. 10 B.

```c
void Bank_Customer ()
{
    // Bank_Customer ();
}

void Deposit_Money ()
{
    // }
}
```

Fig. 10 Notations for representing an actor and its action model from different programming languages.

For the Actor-Action type applications, HAETVE is similar to Use Case approach [4], and is easy to map to Use Case notations as shown in Fig. 11 and Fig. 12.
Fig. 13 Analysis notation mapping from Use Case (UML) to HAETVE.

A process example is shown in Fig. 14 and Fig. 15.

Fig. 14 A process example of Use Case Analysis

With HAETVE, event-responds notation is shown in Fig. 16.

Fig. 16 Event-responds notation with event table in comments

A traditional Activity Diagram can be mapped to a combination of these notations and can be viewed in multiple windows.

Besides Actor-Action diagram, Event-Response diagram, and Activity diagram, all the models can also be automatically generated from a regular program or a legacy software product at all levels. It is a holistic solution.

5.1. Holistic HAETVE Models and diagrams can be generated from an entire software product with its source code written in platform-independent Java language or a platform-dependent programming language to show the product structure, class relationship, and overall static and dynamic properties of the product in Fig. 18 and comparative detailed views of two versions of the product shown in Fig. 19.

Fig. 18 A system level logic chart

Fig. 19 A file level source code version comparison to guide code level understanding within the holistic system view.

5.2. Fully Automatable and Visual

Fig. 20 A query on a model in a system level J-chart.
All models and diagrams can be automatically and completely generated from the source code as shown above in Fig.20. Many models and diagrams of UML are not automatable, such as Use Case diagram in Fig. 21, because it is not yet included in the software systems being developed.

Unlike UML models, all parts of models and diagrams in HAETVE are inside the software system under development. Actors of the Actor-Action relationship diagrams are easy to take requirement validation and verification, and will not affect the program execution as shown in Fig.22, Fig. 23, and Fig. 24.

HAETVE’s colorful screen representation includes model status and interactive response to user without penalty in speed.

5.3. Traceable and Executable models

Fig. 23-25 also show the model is traced back to source code to be able to identify and correct human errors after source code change. The traceability facilitates execution of the correction process.

With HAETVE, when a model is shown, the corresponding model generator is always working and waiting for users’ operation commands through the graphic interface. Users can request corresponding chart generator to show extra information such as the code test coverage, the percentage of the run time spent in each function. Fig. 26 shows how a user selects any function box as a new root to generate a sub-call graph from a J-Chart.

Fig. 26 An application example of the interactive J-Chart for generating a sub-call graph and shows the process to request the model generator to display the location where a runtime error happened (shown with an ‘EXIT’ word added) in the J-Flow diagram in Fig. 27.

Fig. 27 The operation process for displaying the location where a runtime error happened.

If a diagram is better than a thousand words in describing a complex system, then an interactive and dynamic diagram will be much better than ten thousand words to represent a complex system.
6. Transparent-Box Method

As opposed to conventional black box or white box diagnosis method, NSE introduces a transparent testing box to combine functional testing and structural testing seamlessly and dynamically in the entire software development lifecycle. Fig. 28 shows bi-directional testing in every aspect of a software product.

We have the right to be wrong, but we also have the right to be right: NSM makes design become pre-coding, and the coding becomes further design.

7. Conclusion

Existing linear software modeling approaches including MDE, MDA, and MDD are outcomes of linear thinking, reductionism, and the superposition principle that the whole of a complex system is the sum of its components. NSM automating software modeling brings revolutionary changes to software modeling by replacing manual modeling on legacy systems with automating model creation from the legacy source code, studying and improving these working models from the past, and incorporating new requirements into code generation, enabling software coding becomes further design.

NSM has been implemented, and fully supported by product Panorama++. The downloadable Panorama++ for C/C++ on Windows trial version, is on http://www.NSEsoftware.com.

"The next century will be the century of complexity" (Stephen Hawking, January 2000). Opportunities exist in developing platform-independent programming languages which are more suitable for high-level abstractions of a software product; Opportunities are also available in designing better tools to remodel legacy software products written in Cobol, ADA, or FORTRAN programming languages through nonlinear software modeling, etc. In NSM, it can be true, that “The Code is the Design”.[7]

8 References

[7] Jack W. Reeves, http://developers slashdot.org/story/05/03/01/2112257/the-code-is-the-design