Software Coding with NSE

Jay Xiong
NSEsoftware, LLC., USA
jay@nsesoftware.com
www.nsesoftware.com

This paper introduces software coding engineering with NSE on which software coding and software design are combined together closely using the innovated Synthesis Design and Incremental Integration (Growing up) Technique – with NSE software design becomes pre-coding, and software coding becomes further design. The quality of the work products (source code and the documents) is ensured by defect prevention and defect propagation prevention through dynamic testing using the innovated Transparent-box testing method, inspection using traceable documents and the source code, and software visualization.

1. The Problems Addressed

Many useful programming techniques have been proposed by software engineering experts and successfully applied in practices such as Object-Oriented programming technique [Coa93], the Pair Programming technique [Bec99], and the language-specified program editing technique.

But unfortunately, there are still many critical problems exist with today’s software coding engineering paradigm:

(a) it complies with the linear process models and linear software development methodologies based on reductionism and superposition principle, so that almost all software programming tasks are performed linearly, partially, and locally, such as the implementation of program modification, program documentation, test planning, program refactoring, and program version comparison;

(b) it is performed after software design following a top-down order without upstream movement (bottom-up) at all;

(c) it follows the Constructive Holism principle that the components of a software product are completed first, then “Assemble” the whole of the entire product from the components or subsystems, so that system testing and user evaluation is done at or close to the end of the programming;

(d) the coding process and the work products (source code and the documents) are not visible;

(e) the work products are not traceable;

(f) code inspection is performed with separated documents and source code;

(g) modules are coded randomly without systematic ordering support;

(h) there is a need to design and use stub modules to replace the real modules called by the unit being coded in unit testing – but stubs will not return the real value so that it is different from the real application execution;

(i) the quality of the modules coded are low – besides unit testing using stubs, the code inspection is performed inefficiently without support of various traceability, often the modules do not satisfy 100 percent MC/DC (Modified Condition/Decision Coverage)
test coverage [RTCA92]: most software testing tools used for structural testing only offer the capability for statement-level test coverage analysis or branch-level test coverage analysis; although some tools claim that they do support MC/DC test coverage analysis, the test results are shown in textual format without capability to highlight untested condition graphically and directly, so that the test results are hard to review and hard to improve;

(j) it is not supported by a coding-style-independent graphical representation technique and tool, so that the source code written by others is much difficult to read and understand;

(k) there is no a powerful technique and tool to automatically document a source program and make the documents (such as the call graph, the class inheritance chart, the logic diagram of the entire product, and the control flow diagram of the entire product) always consistent to the source code;

(l) the related documents are often inconsistent to the source code after code modification – there is no a systematic way available to update them in time;

(m) programming productivity is hard to calculate without a systematic technique and tool to count the total amount and the percentages of the comment lines, partial comment lines, empty lines, and active lines of the source code;

(n) there is a lack of systematic technologies and tools for the support of re-engineering and reverse engineering to automatically generate huge amount of graphic documents of the program architectures, the program logic of an entire software product and the entire control flow diagram of an entire software product with various traceabilities established, etc.

2. The Solution: Software Coding Engineering with NSE Using the Synthesis Design and incremental integration Technique

The solution is called “NSE-Coding” here which is the application of the innovated Synthesis Design and Incremental Integration (Growing up) Technique in coding engineering. This technique combines software design and coding engineering together to make design become pre-coding and coding become further design.

Here, the Incremental Implementation and Integration means the following activities:

(1) Select one or a set of requirements according to the requirement priority assigned.

(2) From the corresponding call graph (shown in J-Chart notation) of the designed system, highlight the critical module with all modules calling and called by the modules for the selected requirement(s), assign a bottom-up coding order on them.

(3) Perform incremental unit coding according to the assigned order to prevent inconsistency defects between the interfaces of the calling modules and the called modules (see section 7).

(4) Carry out unit testing and integration testing together to remove possible defects through comprehensive testing (including functional testing, structural testing, memory leak and usage violation checking, quality measurement, and performance analysis, etc.).

(5) Recompile the entire program to establish a new version of the program, and then run the program again dynamically.

(6) Different from traditional incremental integration approaches which complete the sub-system design and coding first then carry out integration for the whole system, with NSE the
incremental implementation and integration is done at the same time - each time only one module of the sub-system for the selected requirements will be integrated to establish a new version of the executable program (although different critical paths or different subsystems can be code in parallel, integration should be done by adding one module at a time), so that if something wrong is found, the problems often come from the one added module only rather than the entire sub-system implemented for the selected requirement(s).

(7) Combine the processes of software development, testing, and maintenance together closely through many automated and bidirectional traceabilities for defect prevention in the entire software product development lifecycle.

(8) If some critical problems are found in coding phase, go back to the upper phases to solve the problem – it is possible to give up the previous selected solution method such as in the case that the performance is very bad because the misuse of the virtual memory – in this case, go back to the preprocess to design new solution method.

The major offerings of NSE-Coding are as follows:

(a) it complies with the NSE nonlinear process model and the NSE software development methodologies based on complexity science, so that almost all software programming tasks are performed nonlinearly, holistically, and globally, such as the implementation of program modification, program documentation, test planning, program refactoring, and program version comparison – see Fig. 1 An application example of system-level test planning support through Cyclomatic complexity (the number of decision statements) measurement of an entire software product coded.

(b) with it software design and coding are combined together closely, making design become pre-coding, and coding becomes further design to improve the design – no matter in software design or coding, people are nonlinear, easy to make mistakes and wrong decisions (When a critical issue is found in coding process, the work flow should even go back to the requirement development phase or the preprocess phase) – see Fig. 2 an application example of design becoming pre-coding - to code a module by editing the source code directly from a call graph (shown in J-Chart) generated in the design process;
For instance, in the case that the design shows function A calls function B, but the coding engineers find the function A should call function C and function C should call function B - after coding they can update the design documents by rebuilding the database to make the design result consistent to the code (in this case, they may choose the way to modify the design first, then edit and change the code).

An example of coding becoming further design is shown in Fig. 3 and Fig. 4.
(c) it follows the **Generative Holism** principle that the whole of a software product exists (as an embryo, but executable) earlier than its components, and then grows up with its components to continuously form executable new versions, so that system testing and user evaluation is done at or close to the beginning of the product development;
(d) the coding process and the work products (source code and the documents) are always visible as shown in Fig. 2 and Fig. 5;
Fig. 5 A class shown visually with its logic and control flow

(e) the work products are traceable internally and externally or even dynamically – see section 5.
(f) code inspection is performed with traceable documents and traceable source code – see section 5 too;
(g) modules are coded incrementally with systematic ordering support – see Fig. 6 (ordering for all modules) and Fig. 7 (ordering for a critical path);
(h) in unit testing with capability to control the return value (see section 3 and Appendix C about how to control the return values for a function call statement) there is no need to design and use stub modules to replace the real modules called by the unit being coded.

(i) the quality of the modules coded will be high – besides unit testing in real conditions without using stubs, the quality is ensured through defect prevention and defect propagation prevention supported by dynamic testing using the Transparent-box testing method, semi-automated inspection using traceable source code and documents (the code inspection is performed efficiently with support of various traceabilities - see section 5), software visualization (see section 6), and 100 percent MC/DC (Modified Condition/Decision Coverage) test coverage result support (see section 4);

(j) it is supported by a coding-style-independent graphical representation technique and tools, so that the source code written by others is also easy to read and understand – see
(k) with NSE there is a set of powerful techniques and tools to automatically document a source program – with NSE the source code is also the source for many graphical document generation;

(l) the source code is consistent to the documents after code modification through bidirectional traceability and re-build of the corresponding database.

(m) programming productivity is easy to calculate with a systematic technique and tool to count the total amount and the percentages of the comment lines, partial comment lines, empty lines, and active lines of the source code of an entire software product as shown in Fig. 9;

(n) there are a set of systematic technologies and tools for the support of re-engineering and reverse engineering - to automatically generate huge amount of graphic documents of the program architectures, the program logic of an entire software product, and the control flow diagram of an entire software product with various traceabilities.
established, etc.

(o) it supports parallel coding performed in different subsystems (see Fig. 10) or different critical paths (see Fig. 11), but the integration should still be done by **adding one component at a time** [Bro95-P149] for easy to locate the possible defects. Any time the updated whole product should be executable.

![Fig. 10 Parallel coding for different subsystems](image1)

**Fig. 10 Parallel coding for different subsystems**

![Fig. 11 Parallel coding for different paths](image2)

**Fig. 11 Parallel coding for different paths**

3. **Unit Testing and Integration Testing Support**

With NSE-Coding, unit testing and integration testing are combined together closely using PanoUnit toolset. The interface of PanoUnit is shown in Fig. 12.

Fig. 13 shows the test case generation options.
As described above, with NSE the incremental unit testing is performed without designing and using stubs to replace other units called by the unit being tested, because that according to the incremental coding and testing order, those units called by the unit being tested must have been code and tested already. So, it is a real product testing—when a stub unit is used for the traditional unit testing, it is not a real product testing because the stub unit will not return the real value needed.

Sometimes, we may want a called unit to return some special values for error simulation. It is
supported in two automatic ways - see Appendix C.

The major features of PanoUnit include:
1. Automatically collects the all unit called by the unit being tested together;
2. Semi-automatically creates the test driver to help users complete the driver design.
3. A set of data generation functions are provided for users to choose;
4. It helps users to easily insert assertions for checking the test result;
5. If it is re-testing an existing software product, PanoUnit can collect the all possible values assigned to a global variable or static variable in different locations for users to choose, can also collect the values used to meet the requirement of the constructor of a class object for users to choose;
6. It can compile the program with the driver and the unit as well as the all units called together, and execute the test cases automatically;
7. It can also perform MC/DC test coverage measurement, memory leak checking, etc.;
8. It can show the test results in graphics with the untested branches and conditions highlighted.
9. It can automatically identify whether a test is passed or not.

4. MC/DC Test Coverage Measurement Support

With NSE-Coding, it is strongly recommended to realize 100 percent MC/DC (Modified Condition/Decision Coverage) test coverage for any module in any commercial application and any engineering project, not only for meeting the RTCA/DO-178B level A requirements. Why?

Often people believe that statement-level test coverage is not good enough for the quality assurance of commercial software, but branch-level test coverage may meet the quality assurance requirements. Is it true?

Before answering the question, let’s see some examples.

func1 is a C program module with the source code as follows:

```c
int func1 (int a, int b, int c)
{
    if((a && b && (c==1 || c==11 || c==111 || c==1111 || c==11111))
        return c + c/10 + c/100 + c/1000 + c/10000;
    else
        return 0;
}
```

If we consider branch-level test coverage only, then there are two logic paths; but if we consider MC/DC test coverage, there are eight logic paths as shown in Fig. 14.
func2 is another C program module with the same functionary as func1 but written in different style without using multiple conditions in a decision statement:

```c
int func2 (int a, int b, int c)
{
    if (a)
    {
        if (b)
        {
            switch (c)
            {
            case 1:
                return 1;
            case 11:
                return 12;
            case 111:
                return 123;
            case 1111:
                return 1234;
            case 11111:
                return 12345;
            default:
                return 0;
            }
        }
        return 0;
    }
    return 0;
}
```

The number of source lines of func2 is 25, while the number of source lines of func1 is 8. The number of logic paths for func2 is eight too as shown in Fig. 15.
The source code of a corresponding `main()` program used to test `func1` module and `func2` modules is listed as follows:

```c
void main(int argc, char** argv) {
    printf(" c == %d\n", func1(atoi(argv[1]), atoi(argv[2]), atoi(argv[3])));
    printf(" c == %d\n", func2(atoi(argv[1]), atoi(argv[2]), atoi(argv[3])));
}
```

A simple “Makefile” for running this program is listed as follows:

```makefile
# Makefile
LINK = link
CC = cl

main.exe: main.C
    $(CC) -c main.C
    $(LINK) -out:main.exe -subsystem:console main.obj libc.lib kernel32.lib

clean:
    -erase main.exe
    -erase main.obj
```

After compilation, it is easy to verify that `func1` and `func2` have the same functionality - the execution command lines and the obtained results are as follows:

```
C:\Analyzer1>main 1 1 1
C:\Analyzer1>main 1 1 11
```
To achieve 100 percent branch-level test coverage result for funct module, only two test cases are needed:

The corresponding branch-level test coverage result for func1 is 100 percent tested as shown in Fig. 13 -16.

But if we consider the MC/DC test coverage result, we will find that there are many conditions (and 6 paths) untested as shown in Fig. 17 and Fig. 18.
This result can also be shown clearly in Fig. 18.

The corresponding branch test coverage measurement result of func2 is shown in Fig. 19.
The corresponding MC/DC test coverage measurement result for func2 is shown in Fig. 20 – the untested paths are the same as that for branch-level test coverage measurement.

This result can also be represented clearly in Fig. 21.
For getting 100 percent MC/DC test coverage result for func1, at least 6 more test cases are needed as shown in the following list:

```
main 1 0 1
main 1 1 0
main 1 1 11
main 1 1 111
main 1 1 1111
main 1 1 11111
```

After running those test cases, the MC/DC test coverage result for func1 and func2 are shown in Fig. 22 and Fig. 23 separately.
Fig. 23 All paths in func2 have been tested

How about statement-level test coverage measurement? It is much worse than the branch-level test coverage measurement, so that it should not be used for any commercial software testing at all.

With NSE and the support platform Panorama++/SilverBullet, it is not difficult and not expensive to achieve 100 percent MC/DC test coverage measurement result in unit testing process - see a real application example shown in Appendix B.

**Conclusion:**
(a) A program module written with multiple conditions in a decision statement is much easier to read and understand than that without multiple conditions written in a decision statement.
(b) The size of a program module written without multiple conditions in a decision statement is much bigger than that of a program module written with multiple conditions in a decision statement (in this example, the size of func2 is about 3 times bigger than the size of func1).
(c) As shown in this example, to a program written with multiple conditions in a decision statement, “100 percent (100%) branch-level test coverage” result may be equal to only 20 percent (20%) of the MC/DC test coverage result – it will not be accepted for any commercial or engineering software product: the risk is too high! – in many cases, the execution part of a decision statement with multiple conditions will be much complicated and depended on the conditions, so that if the untested paths in the product development site are executed in the customer site in the real applications, something unexpected may happen to harm the customer’s business.

5. **Semi-automated Inspection Support**

Inspection has been proven a useful technique for finding defects. But traditional inspection is performed using separated documents and source code without automated and self-maintainable
traceability. Differently, with NSE software inspection can be done in a semi-automated way supported by various traceability – see Fig. 24 to Fig. 26.
Fig. 26 An example of external traceability and dynamic traceability among source code and test cases and the related document or program execution

6. Defect Prevention Driven Quality Assurance in programming

As described before, with NSE the quality of a software product being development is ensured with defect prevention and defect propagation prevention through software testing dynamically using the Transparent-box testing method, inspection, and software visualization.

Fig. 27 shows defect prevention through incremental ordering and software visualization: when writing a function call statement for a module being coded (in this example, the module with order number 6) to call a module coded (in this example, the module with order number 4), we can see the diagrammed source code of the called module in a new window to know how many parameters needed and their order to avoid inconsistent defects in writing the calling statement.
In the following sample program, there is a logic defect not easy to find because a program is represented in textual format, and a program with logic defects may execute normally without providing error message but the result could be wrong:

```c
#include <stdlib.h>
#include <stdio.h>
int fun1 (char *s, int m, int n)
{
    int value;
    switch (s[0])
    {
    case '+':
        value = m + n;
        break;
    case '*':
        value = m * n;
        break;
    case '-':
        value = m - n;
        break;
    case '/':
        value = m / n;
        break;
    default:
        value = -1;
    }
    return value;
}
void main(int argc, char** argv)
{
    printf(" The value == %d\n",
            fun1(argv[1],atoi(argv[2]),atoi(argv[3])));
}
```

Some sample test cases may not be able to find the errors such as the following test cases and the results:

```
C:\tem_dir>main + 0 0
The value == 0

C:\tem_dir>main + 2 2
The value == 4
```
But through program visualization, the logic error is much easier to find as shown in Fig. 28.

Fig. 28 Finding logic defects through software visualization

Of course, to this very simple program, we can use more test cases to find it such as the following test cases:

C:\tem_dir>main + 2 3
The value == 6

C:\tem_dir>main * 2 3
The value == 6

But in the real application programs, the execution part for each “case” statement may be complicated, hard to find the defect that a “break;” statement is missing.

7. Quality Measurement for an Entire Software Product and its Each Component

With NSE-Coding, not only the quality of an entire software product will be measured, but any individual modules (units) will also be measured and shown in Kiviat diagram – see Fig. 29 an application example.
For measuring the quality of classes, some special metrics are used, including:

- Lines Of Code per class (LOC)
- Number Of Methods per class (NOM)
- Number of Method Users per class (NMU)
- Weighted Methods per Class (WMC) in multiple complexity metrics
- Depth of Inheritance Tree (DIT)
- Number Of Children per class (NOC)
- Coupling Between Objects (CBO)
- Response For a Class (RFC)
- Lines of Code Reused per class (LCR)
- Ratio of Code Reused per class (RCR)
- Test Coverage per Class (TCC) in multiple test coverage metrics

User can set the standard value through OO-SQA toolkit of Panorama++.

8. Application

NSE-Coding using the Synthesis Design and Incremental Integration Technique has been successfully applied in practices, including the improvement of the NSE support platform Panorama++. All screenshots shown in this paper are come from the application examples of NSE-Coding.

9. The Major Features

The major features of NSE-Coding are briefly summarized as follows:
(a) Holistic – it is based on **Generative Holism** principle
(b) Incremental – coding is done incrementally
(c) Parallel – parallel coding is supported to avoid waiting something
(d) Visual – the coding process and the work products (the source code and the documents) are visible
(e) Traceable – the work products are traceable internally and traceable to the test cases and the documents
(f) Consistent – the design documents and the source code are consistent
(g) Combined with software design together closely
(h) Driven by defect prevention and defect propagation prevention
(i) The source code coded is always executable entirely

10. **Conclusion**

This paper introduced NSE-Coding which applies the innovated **Synthesis Design and Incremental Integration** Technique. With NSE coding is performed holistically and incrementally by complying with the NSE nonlinear process model and the NSE software development methodology based on complexity science. The coding process and the work products (source code and the related documents) are visible. The quality of the coded programs is ensured through defect prevention and defect propagation prevention performed with dynamic testing using the Transparent-box testing method and software visualization plus inspection using traceable documents and traceable source code.

With NSE product design becomes pre-coding, and the coding becomes further design – **source code is the source for generating most graphical software documents - to keep the documents consistent to the program, and traceable to and from the source code.**

**References**