Software Design Engineering under NSE

Jay Xiong
NSEsoftware, LLC., USA
jay@nesoftware.com

In the software design phase the major tasks include the planning of the solution according to the requirement specification, design of the software architecture, design of the data structure, design of the interfaces, design of the algorithms, design of the classes/modules, and design of the documents.

This paper will introduce how software design engineering can be performed holistically, globally, virtually, visually, and efficiently using the innovated NSE Software Development Methodology and Synthesis Design and Incremental growing up (Implementation and Integration) Technique in the software design phase.

1. The Major Problem Addressed

Although in software engineering many methods, techniques, and tools have been developed for software design and applied in practices, there are still many critical problems existing with the old-established software design paradigm because

(a) the old-established software design paradigm (including the methods, techniques, and tools) is based on linear thinking, reductionism, and the superposition principle that the whole of a complex system is the sum of its components, so that almost all software design tasks and activities are performed linearly, partially, locally, and qualitatively – through “Analysis”

(b) it follows the Constructive Holism principle that software components are developed first, then, “Assemble the product from the product components, ensure the product, as integrated, functions properly and deliver the product.” [CMMI1.1]. It handles a software product as a machine which can be assembled, rather than a logic product created by people.

(c) often the designed results consist of many small pieces without a holistic whole for a complex software system.

(d) the designed results are not traceable – hard to review and hard to understand.

(e) even if a holistic result can be designed and shown graphically, without traceability it is still useless because there are too many connection lines, making the designed result hard to view and hard to understand.

(f) the designed results are not directly executable – an Upstream Quality Assurance strategy cannot be implemented dynamically through testing to prevent defects and prevent defect propagation early in the product design phase.

(g) the designed results are hard to update and maintain, no matter if they are represented in text or graphics.

(h) the designed results and the related documents are not traceable backwardly to the requirements or forwardly to the source code.

(i) the designed graphical documents are stored in hard copies or XML or PostScript format, requiring a huge amount of space.

(j) about the design documents, there are two extremes: one is requiring a huge amount of documents but most of which are useless because they are inconsistent with the source
code after code modification performed again and again; another one is based on the concept that “Only the source code is the best document” so that only a few design documents will be provided – making the software product more difficult to maintain.

(k) working with the linear process models, the defects introduced in the design phase easily propagate to the lower phases to make the defect removal cost increase tenfold several times.

(l) the application results of the old-established software design paradigm show that today the software project success rate is still very low - only about 30%.

2. Outline of the Solution for Software Design with NSE

The solution offered by NSE for software design using the innovated **Synthesis Design and Incremental growing up** Technique will be described in detail in this paper later. Here is the outline of the solution:

(a) It is based on nonlinear thinking and complexity science by complying with the essential principle of complexity science, particularly the Nonlinearity principle and the Holism principle that the whole is greater than the sum of its parts, the characteristics and behaviors of the whole emerge from the interaction of its parts and the interaction of it and the environment, so that with NSE almost all the software design tasks and activities are performed holistically and globally.

(b) It complies with the **Generative Holism** principle that the whole of a complex system may exist (as an embryo) earlier than its components, then grows up with its components incrementally as shown in Fig. 1.

(c) The designed results are holistic for the entire product.
(d) The designed results are traceable – easy to review and understand.
(e) With traceability, no matter how complex a software product is, we can easily highlight a module with all the related modules calling and called by it to make the designed results much easier to review and understand.
(f) The designed results are directly executable – an **Upstream Quality Assurance** strategy can be implemented dynamically through testing to prevent defects and prevent defect propagation early in the product design phase.
(g) The designed results are easy to update and maintain - after modifying the source code or the stub programming source code, the design documents can be automatically updated.
(h) The designed results and the related documents are traceable backwards to the requirements or forwards to the source code.
(i) The designed graphical documents virtually exist without storing any hard copies in disk or the computer memory – they are automatically generated though several hash tables virtually.
(j) About the design documents, a huge amount of documents will be automatically generated which are always consistent with the source code – making the software product much easier to maintain.
(k) Working with the NSE nonlinear process models through defect prevention and defect propagation prevention using traceable documents and dynamic testing using the Transparent-box testing method, the defects introduced into a software product and the defects propagated to the maintenance phase will be greatly reduced.
(l) The application results show that working with the NSE process model and the NSE software development methodology, it is possible for the NSE software design paradigm to help software organizations double their product productivity and the project success rate.

3. Description of the Innovated “Synthesis Design and Incremental growing up” Technique

(1) Basic Ideas
   (a) As pointed out by Aristotle, “The whole is more than the sum of its parts.”
   (b) Software is people oriented – people are the first-order components in software development[Co099]
   (c) People are nonlinear
   (d) People make mistakes and wrong decisions often
   (e) So, design and coding should be a two-way process by combining design and coding together closely (Top-down + Bottom-up)
   (f) With NSE design becomes pre-coding, coding becomes further design

(2) What is Synthesis? What is Analysis?

   Synthesis means "to put together" and analysis means "to loosen up" respectively. Analysis is defined as the procedure by which we break down an intellectual or substantial whole into parts or components; Synthesis is defined as the opposite procedure: to combine separate elements or components together to form a coherent whole – “1 + 1 > 2”
According to the Generative Holism principle of complexity science, the whole of a complex system exists earlier than its components – as an embryo, then grows up with its components.

Here, “Synthesis Design and Incremental growing up” means:

(a) Combining all NSE components together and make them work together closely (such as sharing the unique database, using a common interface, etc.) to form the whole of NSE for the requirement implementation including the NSE process model, NSE software development methodology, and particularly the NSE software visualization paradigm and the NSE software testing paradigm.

(b) Combining software design and coding together, supported by the entire NSE paradigm as shown in Fig. 2.

(c) Combining the “Top-Down” design approach and “Bottom-Up” design approach together through two-way iteration.

(d) Combining human-intelligence and computer-computing power together to resolve issues, such as error simulation used for realizing 100% MC/DC test coverage, and getting the class test coverage results from their instances (a class cannot be directly executed).

(e) Combining qualitative research and quantitative implementation together such as the test planning through Cyclomatic complexity (the number of the decision statements) for the entire product and each individual module.

(f) Combining textual description and graphical representation together – generating the graphics directly from the stub source code or the regular source code.

(g) Combining complexity science and reductionism together – “Complexity is by levels.” [Bro95-P211] Sometimes we need to combine their application results as well.

(h) Collecting the information, documents, and data related to the requirements, including the
solution method comparison reports, prototype design and risk estimation reports, test cases and the test results, customer evaluation results, and the documents of the algorithms used, etc.

(i) According to the functional requirement decomposition results plus nonfunctional requirements, updating the executable stub system (the preliminary architectures were designed in the preprocess) through dummy programming.

(j) Testing the designed results dynamically using the Transparent-box approach, and reviewing the result statically using traceable documents and test cases and the source code – even if there is only one top-level dummy module (main()) available and executable with different command-line options (see the Application section).

(e) Removing the defects introduced into the designed dummy system through software visualization and inspection, particularly dynamic testing using the Transparent-box approach.

(f) Performing optimization of the designed dummy system to reduce the coupling degree.

(g) Designing the preliminary data structures (class structures) according to the collected information and data.

(h) Compiling and executing the designed dummy system that maps to the functional requirement decomposition plus the non-functional requirements.

(i) Performing detailed design of the modules.

(j) Working with incremental coding to make the system grow up with new versions of the system executable.

(k) Updating the design results through “Design is pre-coding, and coding is further design.” – for instance, the design shows function A calls function B only, but the coding engineers may find that the function A should call function B and function C – in this case, after coding, they can update the design documents by rebuilding the database to make the design result consistent to the code (they may select to modify the design documents first, then change the code).

(l) If critical issues found, going back to the preprocess to choose a suitable solution method and performing prototyping again.

With NSE, the preliminary design of the whole of the software system is performed in the preprocess through dummy programming using dummy modules based on the result of the function decomposition of the functional requirements and the description of the non-functional requirements.

With NSE, defect prevention and defect propagation prevention will be performed in the entire software development process and the maintenance process using the Transparent-box testing method, plus formal inspection and review using traceable documents and test cases and the source code supported by various traceabilities, plus software visualization.

With NSE, the documents hierarchy is specified with a table using bookmarks to indicate the relationship among the related documents and test cases, which will be used to establish the traceability among all the related documents and test cases and the source code though the execution of the test cases using Time Tags that are automatically inserted into both the test cases and the test coverage database of the source code for data mapping between the test cases and the source code, and some keywords such as @WORD@, @HTML@, @EXCEL@, @PDF@, and @BAT@ written within the test case description to indicate the format of a document, the file path, and the bookmark to open the document from the corresponding location when the document is traced. The @BAT@ keyword is used for dynamic traceability to automatically execute a batch file.
With NSE, the design process and the designed results are visible for static defect prevention and defect propagation prevention. With NSE, the design results are always executable.

(3) Recommendation for Graphic Document Creation/Generation

It is recommended that in most cases, do not spend too much time in drawing design graphics manually or using a graphic editor (draft graphics are good enough to use for review only), because it is time-consuming, costly, not traceable, not executable, hard to change, and hard to maintain. I believe in most applications there is no need to draw a graphic manually or using a graphic editor – in most cases all graphics can be generated automatically through dummy programming or regular source code.

Fig. 3 shows a draft graphic drawn manually.

For obtaining the graphic shown in Fig. 3, the dummy program in C/C++ is very simple:

```c
#include <stdio.h>
void d_routing(){}
void d_placement(){}
```
void g_routing(){}  
void g_placement(){}  
void NEW_EDA()  
{
    g_placement();
    g_routing();
    d_placement();
    d_routing();
}

The corresponding J-Chart generated automatically is shown in Fig. 5.

![Fig. 5 The automatically generated call graph corresponding to Fig; 2](image)

Using dummy programming approach, the same modification is easy to perform:

void d_routing(){}  
void d_placement(){}  
void g_routing(){}  
void g_placement(){}  
void dbs() {}  
void paertitioning() {}

void NEW_EDA()  
{
    g_placement();
    g_routing();
    d_placement();
    d_routing();
    dbs();
    partitioning();
}

The modified call graph automatically generated from the modified dummy source code is shown in Fig. 6.
After changing the module NEW_EDA () to main(), the program can be compiled and executed.

(4) Self-Documenting

For easy to maintain a software product, many kinds of documents can be merged into the source code such as the Sequence Diagram to expose time ordering of events/messages - we can describe the same thing within a program comment such as the use of a formatted table in C/C++ shown as follows:

    /* Time-Event table:
    |
    |   Timing   | t1 | t2 | t3 |
    | Events     | Event1 |     |    |
    |            |       | event2 |    |
    |            |       | event3 |    |
    }|
    |
    |
    |
    |   Timing   | t4 | t5 | t6 |
    | Events     | Event4 |     |    |
    |            |       | event5 |    |
    |            |       | event6 |    |
    }|
    |
    |
    |
    
    */
(5) System Hierarchy Design

Through stub programming, a detailed program hierarchy of a complex software product can be designed step by step as shown in Fig. 7 (a) and Fig. 7 (b)

Fig. 7 (a) System decomposition design
(6) Static defect prevention and defect propagation prevention through traceability

With the old-established software engineering paradigm it is difficult to review a complex program hierarchy shown graphically with many modules connected to each other. With NSE all generated charts and diagrams are traceable for helping users perform static defect prevention and defect propagation prevention as shown in Fig. 8.
(7) **Dynamic defect prevention and defect propagation prevention**

Even if only one top module (the main() function, for instance) is preliminarily designed with some command-line options, we can design a set of test cases to test the module dynamically through different command-line options, then the testing tool using the Transparent-box testing method will establish the automated and self-maintainable traceability among the related documents, the test cases, and the source code for preventing inconsistency defects – see the Application section of this paper.

(8) **Data Structure Design**

Data structure design is one of the most important tasks in software design. With NSE it is also be done through dummy programming. Fig. 9 shows a class inheritance chart of a designed program.

(9) **Detailed Logic Design of the Modules**
With NSE it is recommended to perform detailed module design using J-Diagram. Fig. 10 shows a program design example using the Activity diagram of UML.

Fig. 10 A typical Activity Diagram

A sample programming source code used for representing the corresponding product design specified in the activity diagram is listed as follows:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iomanip.h>
#include "my_include.h"

void main(int argc, char** argv)
{
    user_login_name = get_user_login_name();
    saved_user_password = get_saved_user_password();
    if(!user_registered())
    { /* handling user register. */
        if(saved_user_password!=NULL && user_password!=NULL)
        { /* user Enters Email and Password. */
            while(!strcmp(saved_user_password, user_password))
                /* user Enters Email and Password. */;
            }
        else /* user Enters Email and Password. */;
        is_true =1;
        while(is_true)
        {
            if(there_is_at_least_one_selection=user_enter_search_criteria())
            { /* user processes to check out. */
                system presents selection list;
                user selects a title;
                system presents details about selected book; */
                if(user_want_to_buy_this_book())
                { /* user adds this book to shopping car; */
                    if(user_wants_to_continue_shopping())
                        else break;
                }
                else /* system displays notification; */;
                /* user selects shipping optyions. */
                if(user_uses_same_credit_card())
                { /* user confirms  the card. */
                    else /* user adds new credit card information. */;
                    if(user_wants_to_cancel_the_order())
                    { /* user cancels order process. */
                        exit;
                    }
                    /* user confirms order ;
                    system_returns_confirmation_number. */;
                    break;
                }
            }
        cout << "\n This program has been executed successfully.\n";
        }
```
The included file:

class Customer_passwd {
private:
    char login_name[80];
    char password[80];
public:
    int check_password(char* login_name){}
    int change_password(char* login_name, char* password) {}}

class Customer_infor: Customer_passwd {
private:
    char name[80];
    char shipping_address[80];
public:
    int update_info(char* name, char* shipping_address){}
    /* other member function. */
};

char* user_password;
char* saved_user_password;
char* user_login_name;
int there_is_at_least_one_selection;
int is_true;

int m=1,n=3;
char* p="abc";

int user_registred(){return m;}
char* get_saved_user_password(){return p;}
char* get_user_login_name(){return p;}
int user_want_to_buy_this_book(){return m;}
int user_wants_to_continue_shopping(){return n;}
int the_shipping_address_changed_since_last_order(){return n;}
int user_uses_same_credit_card(){return n;}
int user_wants_cancel_the_order(){return n;}
int user_enter_search_criteria(){return n;}

The J-Diagram generated from the listed dummy programming source code is shown in Fig. 11.
Fig. 11 The generated J-Diagram from the listed dummy source code, shown with the test coverage measurement result after dynamic testing.
4. Application

The following shows an application example for a new EDA software product design.

(a) Complying with Deming’s product quality assurance principles – “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.”[Dem86] – with NSE, the quality of a software product designed is ensured from the first step in top-down product hierarchy design with only the main() program for handling some command-line options – see the dummy source code listed below:

```c
#include <stdio.h>
#include <string.h>

void main(int argc, char** argv)
{
    int ERROR_CODE;
    if(argc != 3 && argc != 4)
        printf("Error found in the command-line.\n\n");
    else if (argc == 3){
        if(strcmp(argv[1],"global_placement")==0)
            // calling g_placement(argv[2]);
        else if(strcmp(argv[1],"global_routine")==0)
            // calling g_routing(argv[2]);
        else if(strcmp(argv[1],"detailed_placement")==0)
            // calling d_placement(argv[2]);
        else if(strcmp(argv[1],"detailed_routing")==0)
            // calling d_routing(argv[2]);
        else if(strcmp(argv[1],"partitionning")==0)
            // calling partitioning(argv[2]);
        else if(strcmp(argv[1],"ordering")==0)
            // calling ordering(argv[2]);
        else
            // calling printf("Invalid name: %s\n",argv[1]);
    } else if (strcmp(argv[2],"dbs_build") == 0)
        // calling dbs_build(argv[2],argv[3]);
    else printf("Error! Invalid name: %s\n",argv[1]);
}
```

Usually, with NSE in the beginning of product design, some documents should be ready for use, including the corresponding requirement specification, the test requirement specification, the prototyping documents, the product development plan, etc., so that according to the test requirement specification and the command-line options (GUI operation options), we can design a corresponding test script file as follows:

```
# An example of using transparent-box to prevent defects in requirement analysis phase
# and initial design phase of a software product development.
# test case 1
# test purpose: to find the inconsistency among all related documents for the global placement sub-system
# The related requirement specification: @HTML@ C:\EN_transparent_box\new_EDA_specifications.html#G_PLACEMENT
# The related prototype design document: @WORD@ C:\EN_transparent_box\prototyping.doc bmname g_placement
# The related development plan: @BAT@ C:\EN_transparent_box\ganttpro6.bat
# The expected execution path:
# [path] main (int, char**) {s0, s2}
# [NOT_HIT] \path [NOT_HIT]
# Expected output : none
# The directory and the execution command:
C:\EN_transparent_box
new_EDA global_placement -dbs=my_dbs
# test case 2
```
After running the 4 test cases, the test coverage result is shown in Fig. 12.
Now it is the time to perform defect prevention and defect propagation prevention through dynamic testing and review using traceable documents and the test cases and the source code:

After running the test cases, we can perform forward tracing and backward tracing to find and remove defects – see Fig. 13 to Fig. 17.

Fig. 13 Tracing the first test case to find the corresponding source code and automatically open the related documents (the requirement specification, the prototyping documents, and the project development plan) - in this operation, no defect was found

Fig. 14 Tracing the second test case with two inconsistency defects found:
(1) The real execution path did not cover the expected execution path main (int, char**) {s0, s3} – segment s3 is highlighted as untested;
(2) The bookmark for opening the global routing section of the prototyping document, g_router, pointed to the wrong section – the global placement section.
Removing the defects:

1. find the location for the first defect and modify the main() program

   From:
   ```c
   else if(strcmp(argv[1],"global_routine")==0)     else if(strcmp(argv[1],"global_routine")==0)
   ; // calling g_routine(argv[2])               ; // calling g_routing(argv[2]);
   ```

   TO:
   ```c
   else if(strcmp(argv[1],"global_routine")==0)     else if(strcmp(argv[1],"global_routine")==0)
   ; // calling g_routine(argv[2])               ; // calling g_routing(argv[2]);
   ```

2. find the mistake related to the bookmark, g_router (Fig. 15), and fix it (Fig. 16):

   Fig. 15 Locating the mistake of the bookmark, g-router
   Fig. 16 Fixing the bookmark mistake

Fig. 17 Verifying that the two defects have been removed through backward tracing from segment s3 (the test case 2 was traced and the related documents were opened without defects)
(b) The designed result after adding the second level is shown in Fig. 18.

![Fig. 18 The design result after adding four subsystems](image)

(c) The top-down design result after adding some more levels is shown in Fig. 19.

![Fig. 19 The top-down design result after adding some more levels](image)

Fig. 20 shows that the designed results are always traceable.

![Fig. 20 Tracing a module (load_dbs() ) to highlight the all related modules](image)
The corresponding dummy programming source code for the main() module is listed as follows:

```c
#include <stdio.h>
#include <string.h>

extern int dbs_build(char*, char*);
extern int d_routing(char*);
extern int d_placement(char*);
extern int g_routing(char*);
extern int g_placement(char*);
extern int partitioning(char*);
extern int ordering(char*);

void main(argc, argv)
int argc;
char **argv;
{
    if(argc != 3 && argc != 4)
    {
        printf("Usage: \n");
        printf("new_EDA dbs_build -conditions=condition_file -dbs=database_file \n\n");
        printf("new_EDA global_placement -dbs=database_file \n\n");
        printf("new_EDA global_routing -dbs=database_file \n\n");
        printf("new_EDA detailed_placement -dbs=database_file \n\n");
        printf("new_EDA detailed_routing -dbs=database_file \n\n");
        printf("new_EDA partitioning -dbs=database_file \n\n");
        printf("new_EDA ordering -dbs=database_file \n\n");
    }
    else if (argc == 3)
    {
        if(strcmp(argv[1],"global_placement")==0)
            g_placement(argv[2]);
        else if(strcmp(argv[1],"global_routing")==0)
            g_routing(argv[2]);
        else if(strcmp(argv[1],"detailed_placement")==0)
            d_placement(argv[2]);
        else if(strcmp(argv[1],"detailed_routing")==0)
            d_routing(argv[2]);
        else if(strcmp(argv[1],"partitionning")==0)
            partitioning(argv[2]);
        else if(strcmp(argv[1],"ordering")==0)
            ordering(argv[2]);
        else
            printf("Invalid name: %s\n",argv[1]);
    }
    else if (strcmp(argv[2],"dbs_build") == 0)
        dbs_build(argv[2],argv[3]);
    else printf("Error! Invalid name: %s\n",argv[1]);
}
```

(d) About the final result of the top-down design of the NEW_EDA system

It is the project I may develop with my colleagues in the future based on complexity science – the existing EDA products for VLSI chip design are completely outdated, because they are outcomes of reductionism and the superposition principle. The NEW_EDA system may have more than fifty thousand function points with more than ten million lines of source code.
5. Making Design Become Pre-coding and Coding Become Further Design

With NSE the design becomes pre-coding: the stub programs used for design can be directly extended to the final source code. For instance, on the designed system structure or call graph, users can click each module-block to edit the stub program and extend it to a final program module as shown in Fig. 21.

Fig. 21 Directly select an editor and then edit a module from a generated call graph

With NSE coding also becomes further design: in the coding process, the software engineers may find something wrong or incomplete and make needed modifications – in this case, the design result will be automatically updated after rebuilding the system database as shown in Fig. 22 and Fig. 23.

Fig. 22 Defects found in coding – two function call statements needed to add

Fig. 23 Updating the design to remove the two defects found
6. The Major Features of the Software Synthesis Design Technique

The major features of the ness (a) It is an engineering approach for software design; (b) It works with the NSE process model and the NSE software development methodology based on complexity science by complying with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle; (c) It complies with the Generative Holism principle of complexity science that the whole of a complex system exists earlier (as an embryo) than its components, and then grows up with its components; (d) It follows the rule that people is the first-order element in software engineering, and the natural law about human beings that people are linear and they easily make mistakes and wrong decisions, so that it combines software design and software coding together to make design become pre-coding and coding become further design; (e) It meets the NSE Upstream Quality Assurance strategy from the first step to the end of the software development lifecycle through defect prevention and defect propagation prevention by dynamic testing using the Transparent-box method, review and inspection using the traceable documents and source code, and software visualization; (f) The work products designed using this technique are holistic, visual, traceable, and always executable; (g) It is a component of the entire NSE software engineering paradigm for efficiently handling the essential issues existing with today’s software development: the complexity, changeability, invisibility, and conformity, defined by Brooks [Bor95-P182];

7. Summary

The old-established software design paradigm works with the linear process models based on reductionism and the superposition principle that the whole of a complex system is the sum of its components, so that with it almost all software design tasks and activities are performed linearly, partially, and locally – through “Analysis”. The obtained work products using the old-established software design paradigm are not holistic, not traceable, not visible, and not directly executable – it means the quality of the product design is hard to ensure.

With NSE software design engineering is performed using the Software Synthesis Design (and Incremental Growing up) Technique working with the NSE process model and the NSE software development methodology based on complexity science by complying with the essential principles of complexity science, particularly the Nonlinear principle and the Holism principle, so that with NSE almost the all software design tasks and activities are performed holistically and globally – through “Synthesis”. The obtained work products are holistic, visible, traceable, and
directly executable for defect prevention and defect propagation prevention mainly using the Transparent-box testing method – it means the quality of the product design is easy to ensure. With NSE design becomes pre-coding, and coding becomes further design.

References