2007

# Experience report on using object-oriented design for software maintenance

Schneidewind, Norman F.

Journal of Software Maintenance and Evolution: Research and Practice, Volume 19, pp. 183-201, 2007
http://hdl.handle.net/10945/44178

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
*J. Softw. Maint. Evol.: Res. Pract.* 2007; **19**:183–201
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.347

**Practice**

# Experience report on using object-oriented design for software maintenance

Norman F. Schneidewind*,†

*Naval Postgraduate School, Pebble Beach, CA, U.S.A.*

**SUMMARY**

**We experimented with modifying the existing object-oriented (OO) design and C++ code of a software reliability model. Our purpose was to assess the efficacy of OO methods for performing maintenance on mathematical software, using a real-world system (NASA Space Shuttle flight software) to illustrate the approach. In this process, we used variants of UML diagrams to modify our design. We found that although a top-down approach to software maintenance is normally a good idea, it was still necessary to modify the design once the realities of what could be accomplished in the C++ code came to light. As reliability and maintenance are intimately related, we developed reliability risk analysis to show how maintenance changes to our design and code could be used to measure risk. Another maintenance enhancement to the design and code is the use of reliability parameter analysis to assess, in the advance of prediction, the reliability of a set of software releases. We believe this is the first evaluation of software maintenance using OO methods. Copyright © 2007 John Wiley & Sons, Ltd.**

*Received 9 February 2007; Revised 8 March 2007; Accepted 2 April 2007*

KEY WORDS: object-oriented software maintenance; software reliability model; NASA Shuttle software

## INTRODUCTION

We used the NASA Shuttle flight software in an experiment to gauge the effectiveness of object-oriented (OO) methods, and the resultant C++ code, for performing enhancive maintenance on a software reliability model that predicts reliability for the Shuttle software. In the experiment, we use hybrid UML diagrams, adapted to mathematical software, for modifying our original reliability model design. In addition, we developed plots of the reliability and risk predictions to both assess the performance of the Shuttle software and to judge the effectiveness of our maintenance changes.

*Correspondence to: Norman F. Schneidewind, Naval Postgraduate School, Pebble Beach, CA, U.S.A.
†E-mail: ieeelife@yahoo.com

The C++ code resulting from the OO maintenance changes and the corresponding UML diagrams are shown in Appendix A.

According to [1], design change requests often affect interfaces. Hence, the connectivity between modules becomes an important consideration. We consider design changes to be one of the most important factors that affect maintenance and we model this effect in our OO maintenance approach to account for the ripple effect of changes at the interface between design and code (e.g., perturbation of changes in UML diagrams to C++ code).

*Caveat.* We do not claim that this analysis is comprehensive. It is essentially one researcher's experience, with reports below on other researchers' views. A comprehensive review of the literature did not unearth reports on this *specific* application of OO methods to mathematical software. Despite this limitation, we feel our results will be of interest to readers who are concerned with the maintenance of non-traditional OO applications (e.g., software reliability models).

## RESEARCHERS VIEWS OF OO SOFTWARE MAINTENANCE

According to [2] and [3], OO design can play an important role in maintenance especially if design-code consistency is maintained. One of our concerns is whether consistency can be maintained from OO design and C++ code in our initial software reliability model through to the maintenance of the system. This involves incorporating reliability parameter evaluation and risk analysis as new components in the system. Parameter analysis (to be explained in more detail later) is a method for using model parameters to estimate the likely reliability of a set of software releases in advance of actual reliability prediction. Risk analysis involves computing a metric, based on time to next failure predictions, of the risk of not meeting reliability requirements.

As stated by Alaga *et al*. [4], one approach is to differentiate between a given modification (primary modification) and the changes resulting from it in the affected part of the system (secondary modifications). They add that before changing the design of a particular object, the maintainer can partition the components of the design into two groups: one consisting of components that will be affected by the design change and the other consisting of components that will be unaffected by the design change. Only the group that is affected by the design change is required to be subjected to further analysis.

We find the distinction between primary and secondary modifications useful because, in our situation, we want to distinguish a primary change at the design level from a secondary change at the code level. For example, by adding risk analysis to our model at the design level, we need to be sensitive to seemingly 'minor' changes at the code level such as declaring the variables that comprise risk! We perform component partitioning by fencing off existing components at the design level in the modified UML diagrams from the added components and by delineating original fragments at the code level from added fragments.

In [5], Eierman and Dishaw suggest that there is a lack of empirical evidence for claims that OO software has an advantage for software maintenance. A field study design, using students, examined the difference in process when using an OO language compared with using a third-generation language. They found a difference in the process used by the two groups of students. Students using the OO language found that they performed more planning and diagnosis activities and students using a third-generation language found that they performed more knowledge building activities.



DOI: 10.1002/smr

This result is interesting because, in our view, planning and diagnosis are to be valued because it is wise to plan maintenance before implementing it, followed by diagnosis to see whether the implementation comports with the plan.

## RESEARCH OBJECTIVES

Our research objectives constitute the problem statement and requirements for performing maintenance.

(1) Investigate the efficacy of the OO paradigm as a model for performing software maintenance.
(2) Determine whether the UML diagrams and C++ code that we have used for developing a software reliability model [6] have the flexibility and modularity for performing maintenance. In particular, we wanted to see whether reliability parameter evaluation and risk analysis could be conveniently added to the software reliability model.
(3) Develop a secure OO design and code because maintenance should not be restricted to changes; they should provide the user with a secure system. For example, data should not be mixed with code because doing so can create avenues for future bugs [7]. We implement this policy in our C++ code. It is not clear whether this provision can be effectively accomplished in UML diagrams. We do not cover external threats to security. Rather, our concern is with the various ways in which changes to the design and code can corrupt the software. The following are some examples:

    (i) data type is not checked on input, leading to corruption of data and computations;
    (ii) reality checks of data and computations are not made, resulting in, for example, unreal software reliability predictions (e.g., time to next failure is predicted to be less than zero);
    (iii) bounds and range checks are not made on data, with resulting exception handling and exit to the operating system.

    A good method for avoiding these problems in C++ code is to use the *assert* command, which will cause the termination of the program if the assert condition is not satisfied. Assertions had their origin in program verification. For the systems developed in industry, construction of assertions and their use in showing program correctness is a near-impossible task. However, they can be used to show that some key properties are satisfied during program execution [8]. We find assertions very useful in ensuring that our program will terminate if, for example, a prediction of time to next failure is less than zero or the failure count used in predictions is not greater than zero.
(4) Use reliability prediction to identify those software products in the NASA Space Shuttle flight software that have the highest reliability and, hence, are the most maintainable, recognizing that reliability and maintainability are intimately related.

## DESIGN OF THE EXPERIMENT

In order to answer the research questions, we used an existing set of UML diagrams and C++ code that were used to develop a software model and modified them to introduce new model features.



DOI: 10.1002/smr

Our aim was to see how adaptable OO design and coding are to maintaining mathematical software; we played the devil's advocate in making this assessment. We do not know of any way to quantify the effectiveness of the adaptation. Instead, we rely on subjective evaluations. For example, what is the benefit of using classes to represent variables and functions to be added to a software reliability model, given the confusion introduced by calling variables attributes and calling function methods? Such an approach *might* be considered excess baggage in attempting to do something simple, such as adding risk analysis computations to an existing software reliability model! Furthermore, in the OO approach we would need to declare the risk analysis class as *public* or *private*. We can use an object to store data in variables and to call functions to perform actions. However, we can already do these things in existing C++ code without incurring the complexity of the OO approach. The claimed advantages of the OO approach [9] were scrutinized for their applicability to mathematical software maintenance.

## CONDUCT OF THE EXPERIMENT

As maintenance is mainly about making changes to existing software, we noted the ease, or lack thereof, of making changes to the existing design and code. A major modification was the addition of reliability model parameter evaluation and risk analysis. Reliability model parameters are quantities computed by statistical analysis to fit the data to the model. In [10], we developed the concept of the ratio of the software reliability parameters $\beta/\alpha$ (see the definitions section below). This ratio provides insight into which software products would have high reliability and those that would have low reliability: the higher the ratio, the higher the reliability. The reason for this is that a high rate of change of a decreasing failure rate coupled with a low initial failure rate tends to be associated with high reliability products. With this information in hand, software engineers could allocate resources optimally to those products needing the most attention (i.e., low reliability products) rather than wasting resources on products that already have high reliability.

In addition, we have found that the risk of software failures is a useful predictive function for indicating whether software is ready to deploy [11]. Thus, this is a second component that we include in our existing model by using OO maintenance.

## DATA USED IN THE EXPERIMENT

Table I shows an example of the type of failure data that was used to develop, test, and maintain the software reliability model. Failure data are needed to estimate model parameters and to test the design. Once the design is complete, the data are also used to test the modified design and code. 'Days from Release' refers to the number of days since release by the contractor to NASA when the failure occurred. This quantity is divided by 30 to obtain 'Days/30' in order to have a convenient time interval for failure counts used in reliability prediction. 'Interval difference' is used in computing failure rate. 'Number of intervals' is the rounded up value of 'Days/30' used as an integer in reliability prediction. 'Reference number' refers to the failure report identification. Under 'Severity', 1N refers to threat to mission safety, 2 refers to major performance degradation, and 3 refers to minor performance degradation. 'Introduced by' refers to the maintenance change request that resulted in the failure.



DOI: 10.1002/smr

Table I. Example NASA Shuttle failure data (operational increment 6).

| Days from release | Days/30 | Interval difference | Number of intervals | Reference number | Severity | Failure date | Release date | Module in error | Introduced by |
|---|---|---|---|---|---|---|---|---|---|
| 136 | 4.533 | | 5 | 54081 | 2 | 02-18-85 | 10-05-84 | PL9 | CR59565 |
| 145 | 4.833 | 0.300 | 5 | 55231 | 2 | 02-27-85 | 10-05-84 | CVHHDA | CR69025 |
| 171 | 5.700 | 0.867 | 6 | 60557 | 2 | 03-25-85 | 10-05-84 | CG3011, XG3011 | CR29886 |
| 223 | 7.433 | 1.733 | 8 | 58909 | 2 | 05-16-85 | 10-05-84 | CG3011, XG3011 | CR29886 |
| 293 | 9.767 | 2.333 | 10 | 63225 | 3 | 07-25-85 | 10-05-84 | SM2OPS, SM4OPS | CR69205 |
| 382 | 12.733 | 2.967 | 13 | 55124 | 2 | 10-22-85 | 10-05-84 | SULUPLIN | CR59565 |
| 525 | 17.500 | 4.767 | 18 | 58764 | 3 | 03-14-86 | 10-05-84 | AIBGPCLO | CR69205 |
| 711 | 23.700 | 6.200 | 24 | 63277 | 3 | 09-16-86 | 10-05-84 | VOKKIP | CR69025 |
| 1355 | 45.167 | 21.467 | 46 | 101361 | 3 | 06-21-88 | 10-05-84 | DMPMMMSG | CR59565 |
| 1748 | 58.267 | 13.100 | 59 | 102796 | 3 | 07-19-89 | 10-05-84 | VXINPUT | CR89025 |
| 1951 | 65.033 | 6.767 | 66 | 104015 | 3 | 02-07-90 | 10-05-84 | VXEERROR | CR69025 |
| 2307 | 76.900 | 11.867 | 77 | 104469 | 3 | 01-29-91 | 10-05-84 | GSESRB | DR57915 |
| 5438 | 181.267 | 104.367 | 182 | 109414 | 1N | 08-26-99 | 10-05-84 | GNPTAC | CR59426F |

## APPLICATION CHARACTERISTICS

The following is a summary of the Shuttle maintenance activity.

- Number of classes modified: 3.
- Number of objects modified: 8.
- Number of methods modified: 6.
- Number of people on project: 200.
- Size of project: approximately 400 000 source lines of code for all software releases.
- Complexity of project: highly complex aerodynamic equations implemented in software.
- Traceability between maintenance and design: CMMI Level 5.

A failure in code is not only traced to individual instructions and product design documents, but also to the process step that caused the failure [12].

## RESULTS FROM THE MAINTENANCE OF THE SHUTTLE SOFTWARE RELIABILITY MODEL

We start with definitions to set the stage for analyzing the results of the software reliability maintenance actions, including the distinction between dependent and independent variables and how the variables are manipulated. The definitions are repeated in the UML diagrams, which are given in Appendix A, so that the reader does not have to refer back to this section.

### Definitions

*Operational increment (OI)*: a Shuttle software release wherein each release includes all of the software from the previous releases plus the increment of the given release (e.g., all releases must contain



DOI: 10.1002/smr

software to support ascent, orbit, and decent plus new functionality such as repairing the Hubble telescope).

### Parameters

The following parameters are used in the prediction equations below [6].

- $\alpha$, failure rate at the beginning of interval $s$.
- $\beta$, negative of the derivative of the failure rate divided by the failure rate (i.e., the rate at which the failure rate changes).
- $t_m$, Shuttle mission duration.
- $s$, starting interval or period for using observed failure data in parameter estimation.
- $X_{s,t}$, observed failure count in the range $[s, t]$.

### Dependent variables

We use the following dependent variables [6].

- $T_f(t)$, time to next failure(s) predicted at time $t$ for a given number of failures $F_t$.
- *Risk*. According to [13], 'risk is a function of: the possible frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity'. We use this broad definition to encompass our specific definition of risk as the probability (i.e., uncertainty) of failures occurring on a release.
- RCM $T_f(t)$, risk criterion metric for *time to next failure* predicted at time $t$.

### Independent variables

We also use the following independent variables [6].

- $t$, test or operational time used in predicting $T_f(t)$.
- $F_t$, given number of failures used in predicting $T_f(t)$.

## PREDICTION EQUATIONS

We now give the prediction equations (i.e., manipulation of variables) [6].

Equation (1.1) is used to predict time to next failure(s). It is also used in the risk criterion metric RCM $T_f(t)$ in (1.2).

$$T_f(t) = [(\log[\alpha/(\alpha - \beta(F_t + X_{s,t}))])/\beta] - (t - s + 1) \tag{1.1}$$

$$\text{RCM } T_f(t) = 1 - (t_m/T_f(t)) \tag{1.2}$$

When RCM $T_f(t) \geq 0$, the risk criterion metric is in the safe region of operational time $t$; otherwise, the metric is in the unsafe region.

Table II captures the security features at the coding level to help ensure that maintenance changes were made without disrupting the existing code of the software reliability model.



DOI: 10.1002/smr

Table II. Model data security features (see the text for object definitions).

| Objects | Security check | Implementation | Comments |
|---|---|---|---|
| $\alpha, \beta, s, X_{st}, t, t_m, F_t, T_f(t)$ | $>0$<br>$\geq 0$ | *Assert* command | Extremely valuable |
| $T_f(t) = [(\log[\alpha/(\alpha - \beta(F_t + X_{s,t}))])/\beta] - (t - s + 1)$ | $\alpha > \beta(F_t + X_{\text{st}})$ | Condition for predicting $T_f(t)$ | C++ code condition |
| $T_f(t)$, RCM $T_f(t)$ | Verification using alternate computations | Comparison between C++ program and Excel | May be small difference in results due to differences in numerical precision |

In Appendix A, we show a series of modified UML diagrams (Figures A1, A2, A3, and A4) to illustrate the maintenance changes at the design level. Note that these diagrams are coded to show the 'before maintenance' in normal type components and the 'after maintenance' components in italics. We also document the results of our OO design and C++ code maintenance activities in the form of several plots. The first plot, Figure 1, shows how the risk criterion metric (referred to earlier) varies with time to next failure, for various Shuttle OIs (i.e., software releases). As we would expect, risk *decreases* (i.e., becomes more positive) as the time to next failure increases (see (1.2)). The reason that risk *increases* for later releases is that later releases include increasing functionality and complexity.

In order to obtain the perspective of risk from the standpoint of a single OI, Figure 2 shows a plot that dramatically indicates the delineation between the unsafe and safe regions. Note the crossover point from the unsafe to the safe region at the mission duration of 8 days.

As mentioned previously, the parameter ratio $\beta/\alpha$ is a good indicator of the relative reliability of various software products. In particular, rather than expending a lot of effort on detailed predictions to see how our products stack up with respect to reliability (and maintainability), we can first compute $\beta/\alpha$ after the parameters have been predicted but before predictions are made. To illustrate this concept, we provide Figure 3. This figure plots failure rate against the parameter ratio for several OIs. Figure 3 indicates that we would allocate the most resources for improving software reliability and maintainability to OI3 and the least to OI8. Now, of course, in order to illustrate the concept we have predicted the failure rate. In practice, we would not make any predictions until after we have performed the parameter analysis; in this case, we would focus our prediction efforts on OI3. Note that while in Figure 1 OI8 has high risk (i.e., low time to next failure), it has high reliability in Figure 3 (i.e., low failure rate). The reason for this is that the prediction quantities are different in the two figures. It is possible for software to have both low time to failure and failure rate predictions. These are related but different reliability functions.

## RESPONSE TO RESEARCH QUESTIONS

(1) With respect to the efficacy of the OO paradigm for maintenance, the results are valid specifically for maintaining mathematical software, using the OO approach, for safety critical applications like



DOI: 10.1002/smr

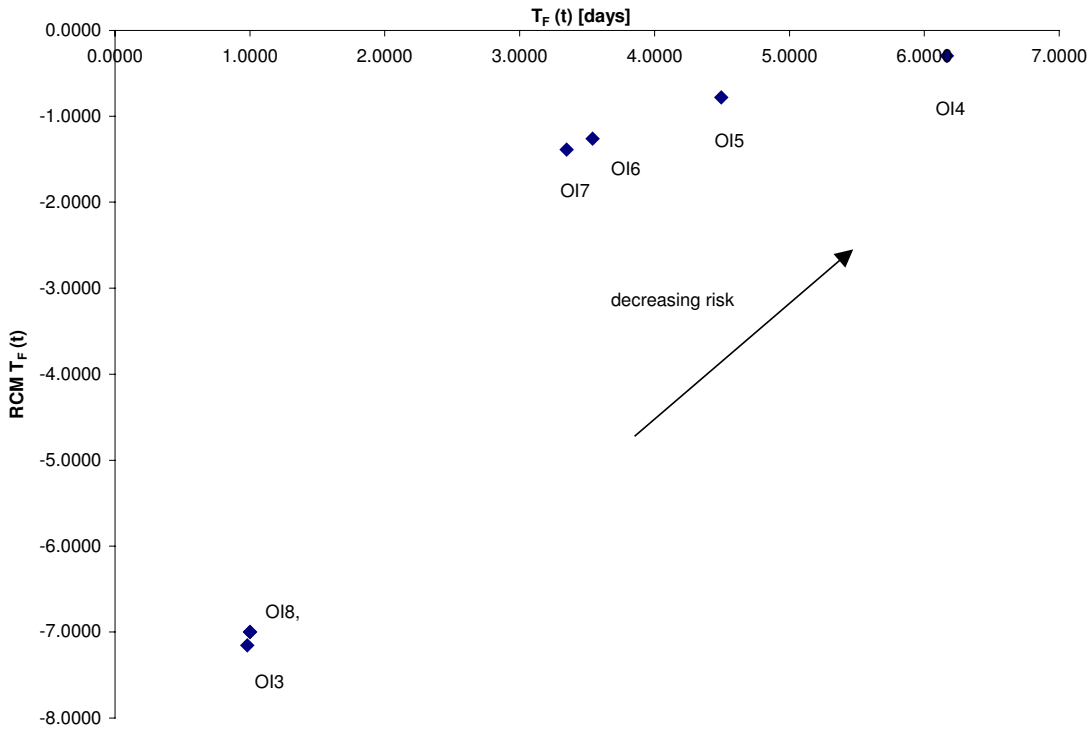



Figure 1. NASA Space Shuttle risk criterion metric RCM $T_f(t)$ versus predicted time to next failure $T_f(t)$ for OIs.

the Shuttle. While the results may be applicable to other software and applications, we do not make this claim.

(2) Concerning whether the modified UML diagrams have the flexibility for making the required maintenance changes to the reliability model, we found that it was easy to use Visio software for creating and modifying UML diagrams. While it was useful, for example, to identify classes, objects, and methods at the design level, using UML diagrams, we did not find it necessary to do so at the code level. The reason for this is that the OO approach was helpful for organizing our thinking about how to make the maintenance changes in the design, but implementing these details in the C++ code would add unneeded complexity that would result in error-prone code.

(3) With respect to developing a secure system, the ability of C++ to fence off code modules and protect them from corruption was very useful. The most useful security aid to maintenance in C++ code is the *assert* capability that causes program termination when the assert condition is not satisfied, such as a prediction equation producing an out-of-bounds result. We found that modified UML diagrams are marginally useful for achieving security by noting security requirements on the charts, but only the code can *implement* the requirements.



DOI: 10.1002/smr

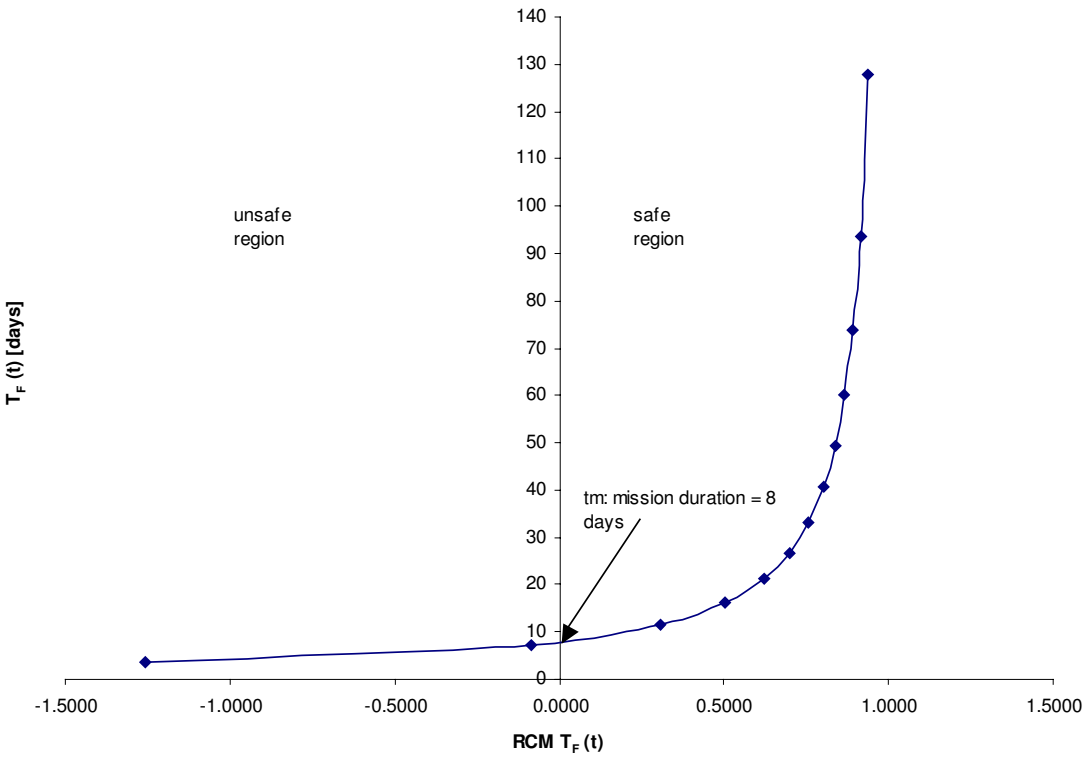



Figure 2. NASA Space Shuttle risk criterion metric RCM $T_f(t)$ versus predicted time to next failure $T_f(t)$ for OI6.

(4) We did integrate reliability with maintainability (see Figure 3), but, counter-intuitively, high reliability and maintainability (e.g., OI8) do not always translate to low risk (e.g., OI8, again; see Figure 1).

(5) Not included in the research questions, but discovered during the research is that despite the conventional wisdom of starting with modifying the design followed by coding, in a linear fashion, did not always prove effective. Sometimes it was necessary to first consider the minutia of the code in order to obtain a correct solution. An example is that originally we introduced a change to the software reliability model to include prediction error, using the actual time to failure in the calculation. Subsequently, in the coding phase, we realized that this quantity could be mistakenly entered as less than zero, causing an erroneous result if inputs were not protected. The *assert* command is useful for providing this protection.



DOI: 10.1002/smr

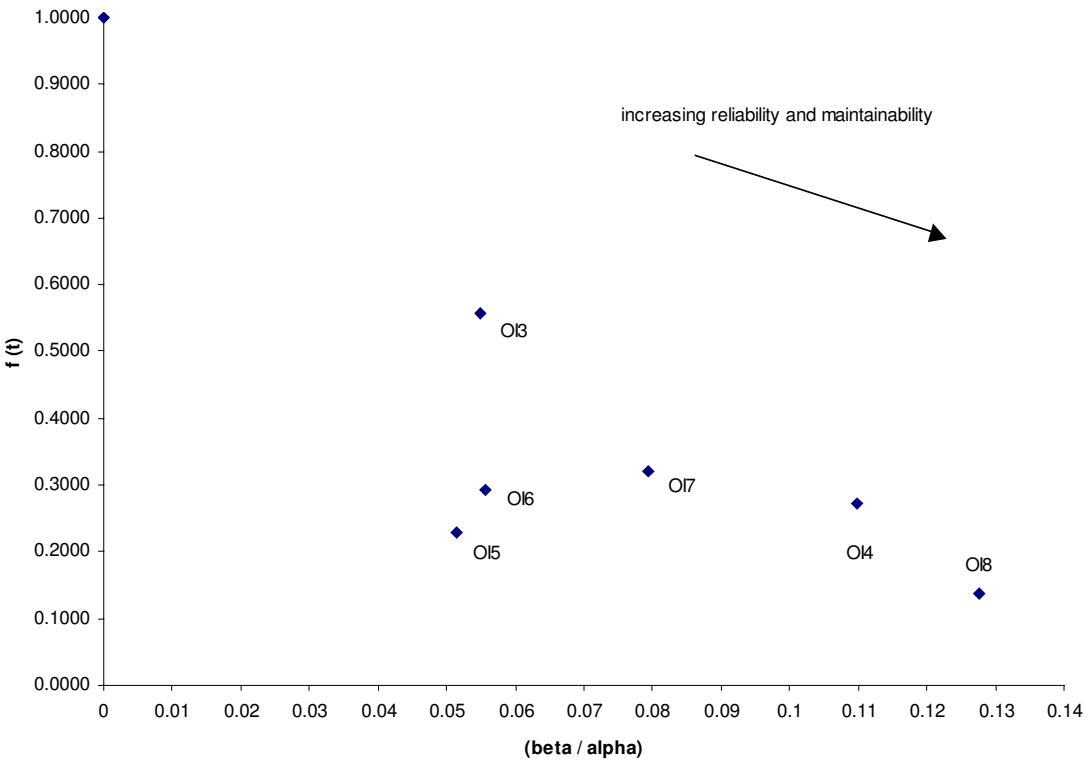



Figure 3. NASA Space Shuttle predicted failure rate $f(t)$ versus parameter ratio $r = (\beta/\alpha)$ for OIs.

## APPENDIX A

```
//Software Maintenance Model with keyboard input and file output
// modified original Software Reliability Program
//s: first failure count interval, t: last failure count interval, ft: specified failure count,
// xst: failure count in range [s,t]
// The time to next failure(s): T = [(log[alpha/(alpha − beta ∗ (ft + xst))]/beta] − (t − s + 1))
// Remaining failures: rm = ((alpha/beta) ∗ (exp(beta ∗ (t − s + 1)))
// Failure rate at time t: fr = alpha ∗ (exp(−beta ∗ (t − s + 1))
// From a security standpoint, do not mix code with data, all data is external to the program
# include <iostream> // specify input output library
# include <math.h> // specify math library
# include <stdio.h>
# include <string.h>
# include <assert.h>
```



DOI: 10.1002/smr

```
using namespace std;
using std::cout; // specify standard screen output
using std::cin; // specify standard screen input
using std::endl; // specify standard end of line

main() // beginning of main program
{       // executable code begins here
FILE *fp;//pointer to type FILE
fp = fopen("c:/models/numbers4.txt", "w");
// "w" opens a text file for writing

double s,t,ft,xst; // declare parameters
double T, fr, rm, alpha, beta; // declare variables T, fr, and r, parameters alpha and beta
double tm, r, ta, e; // added: declare mission duration, risk criterion metric, parameter ratio,
// actual time to failure, and time to failure error

int i, j, k, m, n, p, q, f, ds ; // declare output count, array indices, data set count
j = 0, k = 0, m = 0, n = 0, p = 0, q = 0, f = 0 ; // initialize array indices
i = 1; // initialize output count
ft = 1; // initialize failure count

double arrayrcm [20]; // declare risk criterion metric array (added)
double arrayft [20]; // declare failure count array
double arrayt [20]; // declare time to next failure array
double arrayta [20]; // declare actual time to next failure (added)
double arraye [20]; // declare error array (added)
double arrayrm [20]; // declare remaining failures array (added)
double arrayf [20]; // declare failure rate array (added)
cout << endl; // start output on a new line

cout << "input number of data sets ="; // tell user to input number of data sets
cin >> ds ; // inputted ds
void assert (ds > 0);

cout << "input alpha =" ; // tell the user to input alpha
cin >> alpha; // inputted alpha

// data validity checks, if conditions not met, program terminates
void assert (alpha > 0);
cout << "input beta =" ; // tell the user to input beta
cin >> beta; // inputted beta
void assert (beta > 0);
cout << "input s ="; // tell user to input s
cin >> s; // inputted s
```



DOI: 10.1002/smr

```
void assert (s > 0);
cout << "input xst =" ; // tell the user to input xst
cin >> xst; // inputted xst
void assert (xst > 0);
cout << "input t =" ; // tell the user to input t
cin >> t; // inputted t
void assert (t > 0);
cout << "input tm ="; // tell the user to input mission duration
cin >> tm; // inputted tm (added)

void assert (tm > 0);

if ((alpha > (beta * (ft + xst)))) // test for solution
{
cout << "can obtain solution";

}
else
{
cout << "cannot obtain solution";
}

while (i <= ds)// predict T, if more ds inputs
{// start of while command
cout << endl; // start output on a new line
cout << "data set #:"; // tell user data set number
cout << i ; // output data set number
cout << "input ft =" ; // tell the user to input ft
cin >> ft; // inputted ft

void assert (ft > 0);
// data validity check, if condition not met, program terminates
cout << "input ta ="; // tell user to input actual time to next failure, must enter 0 when null input
cin >> ta; // inputted ta (added)

void assert (ta >= 0);
// data validity check, if condition not met, program terminates
// note: because of calendar dates used for time to failure, it is possible for ta = 0, thus a limit of .1 is
used
T = ((log(alpha/(alpha - beta * (ft + xst)))/beta) - (t - s + 1)); // predict T

rm = ((alpha/beta) * (exp(beta * (t - s + 1)))); // predict remaining failures (added)
void assert (rm >= 0); // condition for valid prediction, if condition not met, program terminates
```



DOI: 10.1002/smr

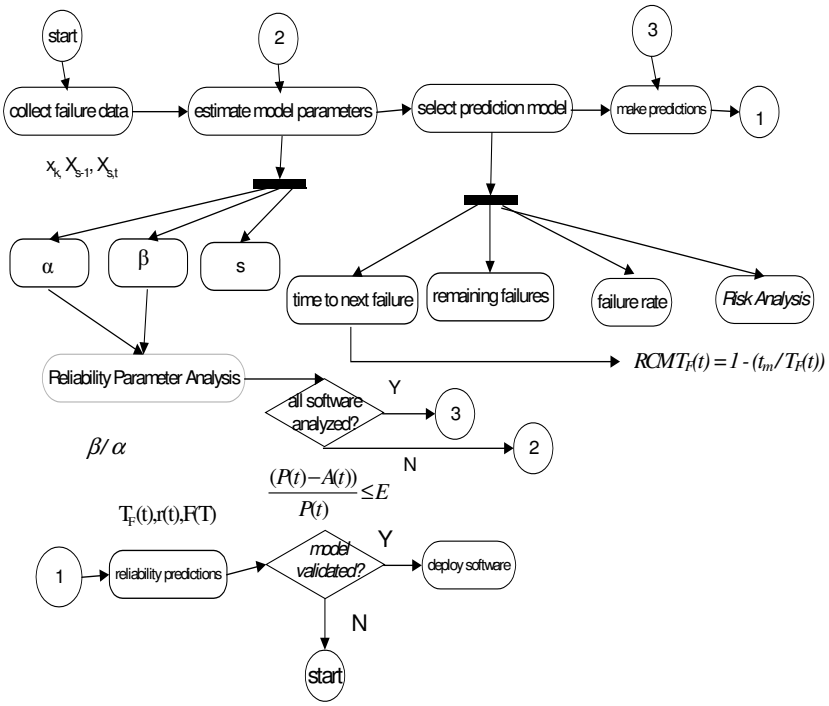
```
fr = (alpha * (exp(−beta * (t − s + 1)))); // predict failure rate (added)
void assert (fr >= 0); // condition for valid prediction, if condition not met, program terminates

arrayt [k] = T; // store T in array
arrayta [m] = ta; // store ta in array (added)
arrayft [j] = ft; // store ft in array
arrayrcm [n] = (1 − (tm/arrayt[k])); // Predict risk criterion metric (added)
arrayrm [q] = rm; // store rm in array
r = beta/alpha; // compute parameter ratio (added)
arraye [p] = ((arrayt[k] − arrayta[m])/arrayt[k]); // compute prediction error (added)
arrayf [f] = fr; // store fr in array
fprintf (fp,"%f%c%\n", r, (char) 6); // output r to numbers3.txt file

fprintf (fp,"%f%c%\n", arrayrcm [n], (char)6); // output rcm to numbers4.txt file

fprintf (fp,"%f%c%\n", arraye [p], (char)6); // output e to numbers4.txt file

fprintf (fp,"%f%c%\n", arrayft [j], (char)6); // output ft to numbers4.txt file

fprintf (fp,"%f%c%\n", arrayt [k], (char)6); // output T to numbers4.txt file

fprintf (fp,"%f%c%\n", arrayta [m], (char)6); // output ta to numbers4.txt file

fprintf (fp,"%f%c%\n", arrayrm [q], (char) 6); // output rm to numbers4.txt file

fprintf (fp,"%f%c%\n", arrayf [f], (char) 6); // output fr to numbers4.txt file

i = i + 1; // increment ft input count
j = j + 1 ; // increment array index
k = k + 1; // increment array index
m = m + 1; // increment array index
n = n + 1; // increment array index
p = p + 1; // increment array index
q = q + 1; // increment array index
f = f + 1; // increment array index
}// end of while command

fclose(fp); // close numbers4. txt file
return 0; // return to the operating system
} // executable code ends here
```

### A.1. Modified and non-standard UML diagrams

At the outset, please note that the purpose of the diagrams is not slavish adherence to all of the details of standard UML diagrams. Rather, the purpose is to *adapt* the diagrams to this *particular* problem



DOI: 10.1002/smr




Definitions:

$T_F(t)$ Time to next failure(s) predicted at time $t$

— r(t) Remaining failures predicted at time $t$

$t_m$: mission duration

E: error threshold

F (T): Cumulative failures predicted at time T

α Failure rate at the beginning of interval s

β Negative of derivative of failure rate divided by failure rate

s Starting interval for using observed failure data in parameter estimation

A (t) actual value of a software reliability quantity

P (t) predicted value of a software reliability quantity

$x_k$ — number of observed failures in interval $k$

$X_{s-1}$ — observed failure count in the range $[1, s-1]$

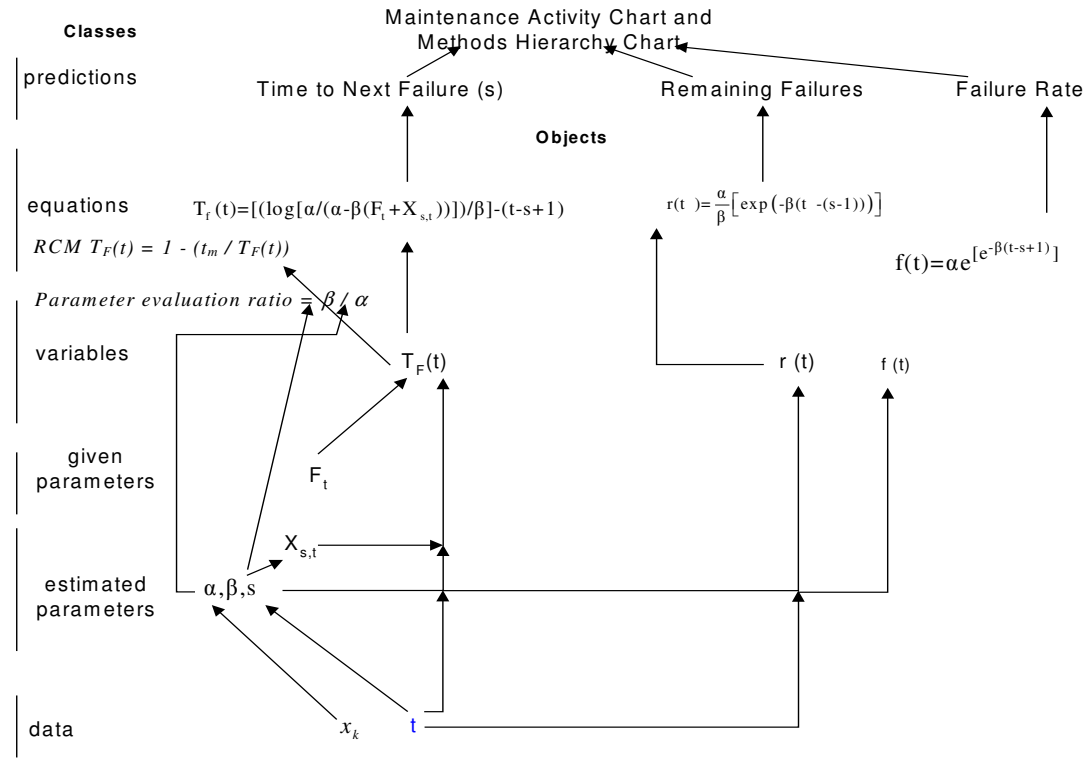
$X_{s,t}$ — observed failure count in the range $[s, t]$

$RCMT_F(t)$: risk criterion metric

Figure A1. Software reliability model maintenance activity chart. New components are shown in italics.



DOI: 10.1002/smr




Definitions:

| | |
|---|---|
| r(t) | Remaining failures predicted at time *t* |
| $T_F(t)$ | Time to next failure(s) predicted at time *t* |
| α | Failure rate at the beginning of interval s |
| β | Negative of derivative of failure rate divided by failure rate |
| s | Starting interval for using observed failure data in parameter estimation |
| t | Last interval of observed failure data |
| T | Time of prediction |
| $x_k$ | number of observed failures in interval *k* |
| $X_{s-1}$ | observed failure count in the range $[1, s-1]$ |
| $X_{s,t}$ | observed failure count in the range $[s, t]$ |

f (t): failure rate

*RCM $T_F(t)$:* *risk criterion metric for time to next failure at time t*

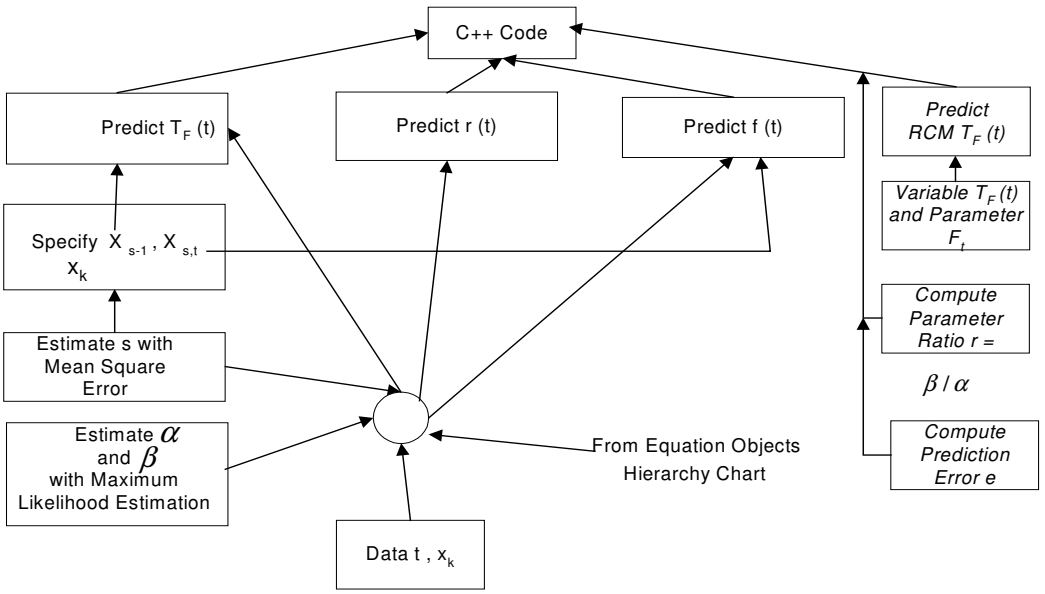
«requirement»
r (t) , TF (t), f (t): floating point and 6 charaters
alpha and beta: floating point and 6 charaters
s, t, floating point and 6 characters
xk, Xs,t: floating point and 6 charaters

*$t_m$: mission duration*

Figure A2. Equation classes and objects maintenance hierarchy chart. New components are shown in italics.



DOI: 10.1002/smr




Definitions:

— $T_F$ (t) Time to next failure(s) predicted at time $t$ f (t): failure rate

— r Remaining failures predicted at time $t$

F (T) Cumulative Number of Failures after time T

$X_{s-1}$ observed failure count in the range $[1, s-1]$

$X_{s,,t}$ observed failure count in the range [s, t]

— s Starting interval for using observed failure data in parameter estimation

α Failure rate at the beginning of interval s

β Negative of derivative of failure rate divided by failure rate

t Cumulative time in the range $[1, t]$

T Time of prediction

$x_k$ Number of observed failures in interval k

*RCM $T_F$ (t) Risk Criterion Metric*

Figure A3. Maintenance methods hierarchy chart. New components are shown in italics.



DOI: 10.1002/smr

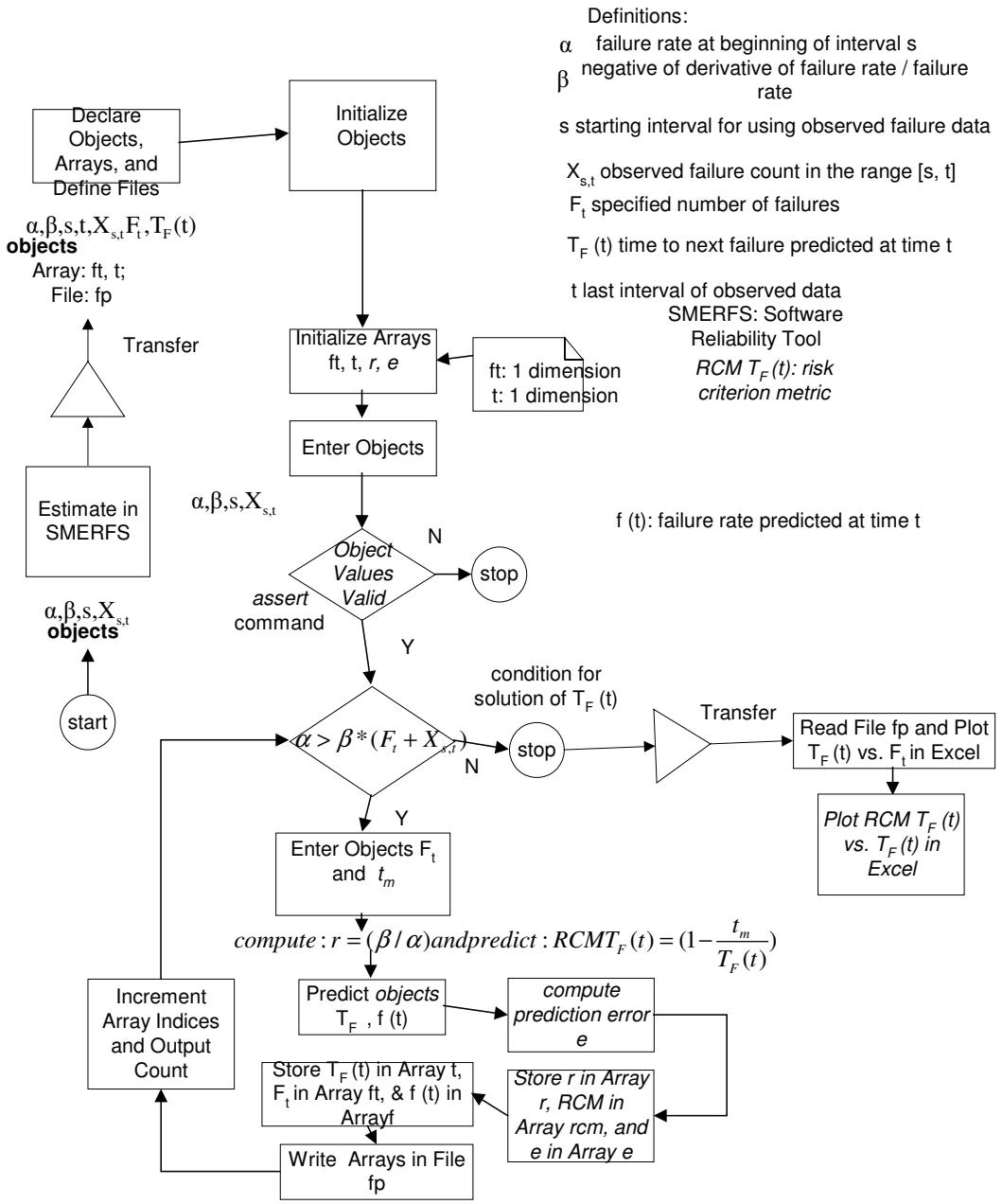



Figure A4. Structure chart of the software maintenance model. New components are shown in italics.



DOI: 10.1002/smr

in maintenance for effectively communicating the problem design. Thus, the basic UML diagrams, created with Visio, were *modified* because they did not convey all of the characteristics of mathematical software development. The diagrams above show several predictions: time to next failure, remaining failures, and failure rate for the Shuttle. Note that on all of these diagrams the new components are coded in *italics*.

In Figure A1, the maintenance activities are depicted that would be necessary, using the OO approach, to change and modify the original software reliability model components to include parameter analysis, risk assessment, and model validity assessment. A maintenance organization would use this diagram to develop the overall plan of activities necessary to make maintenance changes.

Then, in Figure A2, we look at how the classes (data, parameters, and variables) and objects (instances of classes) of the software reliability model feed into equations objects and how the latter are linked to the activity chart (Figure A1) and methods chart (Figure A3). Again, we note the maintenance changes to the model. Maintainers would use this chart to identify the objects to be maintained and their relationships.

The main message of Figure A3 is the identification of the methods (i.e., software reliability prediction equations) and the relationship of the methods with the objects identified in Figure A1. This figure highlights how the parameters, variables, and equations are integrated with maintenance methods, cumulating in directing maintenance changes to linkage with C++ code. A maintenance activity would use this chart to integrate methods and objects prior to writing code.

Finally, Figure A4 shows the interaction of objects with the outside world, namely SMERFS (a software reliability tool used for parameter estimation) and Excel for generating prediction plots and for validating the C++ code predictions.

## REFERENCES


1. Khoshgoftaar TM, Szabo RM. Predicting software quality, during testing, using neural network models: A comparative study. *International Journal of Reliability, Quality and Safety Engineering* 1994; **1**(3):303–319.
2. Antoniol G, Caprile B, Potrich A, Tonella P. Design-code traceability for object-oriented systems. *Annals of Software Engineering* 2000; **9**(1–4):35–58.
3. Fiutem R, Antoniol G. Identifying design-code inconsistencies in object-oriented software: A case study. *Proceedings International Conference on Software Maintenance (ICSM1998)*. IEEE Computer Society: Washington DC, 1998; 94–102.
4. Alaga VS, Qiaoyun RL, Ormandjieva OS. Assessment of maintainability in object-oriented software tools. *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*. IEEE Computer Society: Washington DC, 2001; 194.
5. Eierman M, Dishaw M. The process of software maintenance: A comparison of object-oriented and third-generation development languages. *Journal of Software Maintenance and Evolution: Research and Practice* 2007; **19**(1):33–47.
6. Schneidewind N. Reliability modeling for safety critical software. *IEEE Transactions on Reliability* 1997; **46**(1):88–98.
7. Howard M, LeBlanc DE. *Writing Secure Code* (2nd edn). Microsoft Press: Redmond WA, 2002; 800 pp.
8. Satpathy M, Siebel NT, Rodríguez D. Assertions in object oriented software maintenance: Analysis and a case study. *Proceedings International Conference on Software Maintenance (ICSM2004)*. IEEE Computer Society: Washington DC, 2004; 124–135.
9. Ullman L, Signer A. *C++ Programming*. Peachpit Press: Berkeley CA, 2005; 528 pp.
10. IEEE Reliability Society. Draft standard for software reliability prediction. *IEEE/AIAA P1633$^{TM}$/Draft 5.7*, Software Reliability Engineering Working Group, Definitions and Standards Committee, IEEE Reliability Society, January 2007.
11. Schneidewind N. Predicting risk as a function of risk factors. *Innovations in Systems and Software Engineering* 2005; **1**(1):63–70.




DOI: 10.1002/smr

12. Keller T, Schneidewind NF. A successful application of software reliability engineering for the NASA Space Shuttle. *Proceedings The Eigth International Symposium on Software Reliability Engineering (ISSRE97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 112–113.
13. National Aeronautics and Space Administration. Software safety. *NASA Technical Standard NASA-STD-8719.13A*, NASA, 15 September 1997. Available at: http://satc.gsfc.nasa.gov/assure/nss8719_13.html [1 May 2007].

**AUTHOR'S BIOGRAPHY**

**Norman F. Schneidewind** is Professor Emeritus of Information Sciences in the Department of Information Sciences and the Software Engineering Group at the Naval Postgraduate School. He is now performing research and publishing in software reliability and metrics with his consulting company Computer Research. He is a Fellow of the IEEE, elected in 1992 for 'contributions to software measurement models in reliability and metrics, and for leadership in advancing the field of software maintenance'. In 2001, he received the IEEE 'Reliability Engineer of the Year' award from the IEEE Reliability Society. In 1993 and 1999, he received awards for Outstanding Research Achievement by the Naval Postgraduate School. He was selected for an IEEE U.S.A. Congressional Fellowship for 2005 and worked with the Committee on Homeland Security and Government Affairs, United States Senate, focusing on homeland security and cyber security.



DOI: 10.1002/smr