NASA Contractor Report 198412

NASA-CR-198412
19960008042

Software Safety Progress in NASA

Charles F. Radley
Raytheon Engineers and Constructors
Brook Park, Ohio

October 1995

Prepared for
Lewis Research Center
Under Contract NAS3–26764

National Aeronautics and
Space Administration

Software Safety Progress in NASA

Charles F. Radley
Raytheon Engineers and Constructors
2001 Aerospace Parkway
Brook Park, Ohio 44142
Tel (216) 977-1492
Internet: charles.radley@lerc.nasa.gov
Fax: (216) 977-1495.

Abstract:

November 1995 is the scheduled publication date for the NASA Guidebook for Analysis and Development of Safety Critical Software. Development of the guidebook has substantially focused the thinking of the NASA Software Assurance community with respect to high risk/high value software applications. The guidebook has been developed as a practical "how to" guide, to assist in the implementation of the recent NASA Software Safety Standard NSS-1740.13 which was released as "Interim" version in June 1994, scheduled for formal adoption late 1995.

The Guidebook is in four main parts:

Section 2) System safety context
Section 3) Software Safety planning
Section 4) Development of Safety Critical Software
Section 5) Analysis of Safety critical software.

In addition there is an extensive glossary, appendices and list of references.

Each section is subdivided into a section for each of the following software lifecycle phases: concept, requirements, architectural design, detailed design, and implementation.

Two complementary philosophies were adopted, a) elimination/reduction of faults/errors, and b) fault tolerant techniques.

Both techniques are essential, because it is impossible to eliminate all faults and errors, so some degree of fault tolerance will always be required. However, fault tolerance and redundancy is expensive to implement, so elimination and reduction of faults should be attempted to avoid unnecessary redundancy.

The earlier a fault is corrected in the lifecycle, the lower the cost. Faults identified late are expensive to correct. So emphasis is given to correctness of requirements prior to their implementation in design The most rigorous method of developing requirements is "Formal Methods" which are discussed in some depth in an Appendix. However, Formal

Methods are expensive to implement, so discussion is provided of less costly but less rigorous techniques, and when it is appropriate to use them. Formal Methods requires a substantial investment in training, and procurement of logical tools. It translates traditional human language (e.g., English) specifications into "High Order Logic" (HOL) lemmas The HOL representation reveals defects such as ambiguity, contradictions, double meanings, circular definitions and missing requirements.

Fault tolerance requires hardware redundancy, either parallel paths for "Must-work Functions" (MWFs) or series inhibits for "Must-Not-Work Functions" (MNWFs). To avoid common cause faults, the redundant paths or inhibits must be independent, using dissimilar redundancy. For software to be independent, N-version programming should be used, but this is expensive Different types of N-version programming protect against different types of faults

Analysis techniques include Hazards Analysis, Fault Trees, Petri Nets, Dynamic Flowgraphs, and Resources Guidance is provided on software design techniques and practices.

Software safety costs money, and its value is poorly understood by financial managers. The Guidebook provides guidelines for assessing how much of a program's resources should be devoted to Software Safety as a function of the risk of system failure. These guidelines are intended to assist decision makers to make an informed risk versus cost assessment. Subsequently it will assist software developers and safety analysts to achieve and verify the most appropriate degree of software safety.

Table 1. Guidebook Table of Contents

Table of Tables

TABLE OF FIGURES

1. INTRODUCTION

It is impossible to provide a complete synopsis of the 200 page guidebook in a single conference paper Instead, we have selected a few key technical elements which are the most significant recommendations for software developers and safety analysts

1.1 Scope

The NASA Guidebook for Safety Critical Software - Analysis and Development, was prepared by the NASA Lewis Research Center, Office of Safety and Mission Assurance, under a Research Topic (RTOP) task for the National Aeronautics and Space Administration. The NASA Software Safety Standard NSS 1740 1 prepared by NASA HQ addresses the "who, what, when and why" of Software Safety Analyses. This Software Safety Analysis Guidebook addresses the "how to" The focus of this document is on analysis and development of safety critical software The guidebook can also be used for analysis and development of firmware which is software residing in non-volatile memory, e g., ROM or EPROM

There are many different techniques described in the literature. Here they are brought together, evaluated, and compared The guidebook addresses the value added versus cost of each technique with respect to the overall software safety goals.

1.2 Purpose

The purpose of the guidebook is to provide an aid to the various organizations involved in the development and assurance of safety critical software.

1.3 Acknowledgments

The material presented in the guidebook has been based on a variety of sources These sources are too numerous to list here.

A special acknowledgment is owed to the NASA/Caltech Jet Propulsion Laboratory of Pasadena, California, whose draft "Software Systems Safety Handbook" has been used verbatim or slightly modified in several sections of the guidebook.

We also thank the American Society of Safety Engineers for permission to reproduce portions of the paper. Gowen, Lon D. and Collofello, James S "Design Phase Considerations for Safety-Critical Software Systems". PROFESSIONAL SAFETY, April 1995.

2. System Safety Program

A system safety program is a prerequisite to performing analysis or development of safety critical software

It is often claimed that "software cannot cause hazards", however this is only true where the software resides on a non-hazardous platform and does not interface with any hazardous hardware

2 1 Preliminary Hazards Analysis (PHA)

The PHA is the first of a series of system level hazards analyses, whose scope and methodology is described in the NASA NHB 1700 series documents, and NSTS 13830 Implementation Procedure for NASA Payload System Safety Requirements.

3 Software Safety Planning

This section discusses the level of effort for both software development support tasks, and software analysis tasks to be performed by software development personnel, and software safety personnel respectively. On the development side, the software safety engineer flows safety requirements to the software developers and monitors their implementation On the analysis side, the software safety engineer analyses software products and artifacts to identify new hazards and new hazard causes to be controlled. The analysis and development tasks follow the software development
lifecycle.

The level of effort required is related to the system risk index, based on severity and probability of occurrence of hazards.

Table 3-4 Software Requirements Phase through Table 3-10 Software Module Testing are modifications of tables that appear in the International Electrotechnical Committee (IEC) draft standard IEC 1508, "Software For Computers In The Application Of Industrial Safety-Related Systems". Their set of tables is the best known (but unpublished) planning guide in existence for software safety.

LifeCycle Phase	Tasks and Priorities	How to: Development Tasks	How to· Analysis Tasks
Concept Initiation	Table 3-4 Software Requirements Phase	Section 4 1	Section 5.1
Software Requirements	Table 3-4 Software Requirements Phase	Section 4 2	Section 5 1
Software Architectural Design	Table 3-5 Software Architectural Design	Section 4 3	Section 5.2
Software Detailed Design	Table 3-6 Software Detailed Design Phase	Section 4 4	Section 5 3
Software Implementation	Table 3-7 Software Implementation Phase	Section 4 5	Section 5 4
Software Test	Table 3-8 Software Testing Phase Table 3-9 Dynamic Testing Table 3-10 Software Module Testing	Section 4.6	Section 5 5

4 Safety Critical Software Development

Software safety activities which should be incorporated into the software development phases of a project.

4.1 Software Concept and Initiation Phase

For most NASA projects this lifecycle phase involves system level requirements and design development.

4.2 Software Requirements Phase

The cost of correcting software faults and errors escalates dramatically as the development life cycle progresses. Thus it is important to correct errors and implement correct software requirements from the very beginning However it is generally impossible to eliminate all errors Hence two goals or philosophies are continuously required·

1) Development of complete and correct requirements and correct code

2) Development of fault-tolerant designs, which will detect and compensate for software faults "on the fly".

Both these thought processes must begin during initial requirements development.

4.2.1 Development of Software Safety Requirements

Software safety requirements are obtained from several sources, and are of two types, generic and specific The generic category of software safety requirements are derived from sets of requirements which are commonly used in different programs and environments to solve common software safety problems Specific software safety requirements are system unique functional capabilities or constraints which are identified in three ways:

1) Through top down analysis of system design requirements

2) From the PHA

3) Through bottom up analysis of design data

4 2 2 Generic Software Safety Requirements

Sources of generic software safety requirements

NSTS 19943 Command Requirements and Guidelines for NSTS Customers

STANAG 4404 (Draft) NATO Standardization Agreement (STANAG) Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems

EWRR (Eastern and Western Range Regulation) 127-1, Section 3.16 4 Safety Critical Computing System Software Design Requirements.

AFISC SSH 1-1 System Safety Handbook - Software System Safety , Headquarters Air Force Inspection and Safety Center.

EIA Bulletin SEB6-A System Safety Engineering in Software Development (Electrical Industries Association)

NASA Marshall Space Flight Center (MSFC) Software Safety standard

Underwriters Laboratory - UL 1998 Standard for Safety - Safety-Related Software, January 4th, 1994

4.2 2.1 Fault Tolerance/Independence

Most NASA space systems employ failure tolerance to achieve an acceptable degree of safety. This is primarily achieved via hardware, but software is also important, because improper software design can defeat the hardware failure tolerance.

"Must-Work Functions" (MWFs) achieve failure tolerance through independent parallel redundancy. For parallel redundancy to be truly independent there must be dissimilar software in each parallel path Software can be considered "dissimilar" if N-Version programming is used. N-version programming is discussed below in Section 4 3.1.1 N-Version Programming.

"Must-Not-Work Functions" (MNWFs) achieve failure tolerance through independent multiple series inhibits For series inhibits to be considered independent they must be generally controlled by different processors containing dissimilar software.

4 2 2 2 Hazardous Commands

4 2 2 3 Coding Standards

One class of generic software requirements are coding standards, these are in practice "safe" subsets of programming languages. These are needed because most compilers can be unpredictable in how they work For example, dynamic memory allocation, the defaults chosen by the compiler might be unsafe See 4 5 Software Implementation.

4.2 2.4 Timing, Sizing and Throughput Considerations

System design should properly consider real-world parameters and constraints, including human operator response times, and control system response times, and flow these down to software appropriately. Adequate margins of capacity should be provided for all these critical resources.

Automatic safing is often required if the time to criticality is shorter than the realistic human operator response time, or if there is no human in the loop

Control system design should be based on the established body of control theory, such as dynamic control system design, and multivariable design in the s-domain for analog continuous processes Sampled analog processes should make use of Z-transforms to develop difference equations to implement the control laws This will also make most efficient use of real-time computing resources

Quantization: Digitized systems should select wordlengths long enough to reduce the effects of quantization noise to ensure stability of the system.

4 2.3 Formal Methods - Specification Development

Formal Methods is a process which translates all requirements into a quasi-mathematical language of logical expressions. This forces a singular interpretation of all the requirements, and makes it easier to find missing, incomplete or conflicting/inconsistent requirements. This ensures that the specification analysis is thorough, accurate, and consistent Ad hoc specification analysis is unlikely to screen all the requirements errors, except for relatively simple systems However, Formal Methods are expensive to implement and require a substantial investment in training in order to be effective, so they are not appropriate for low risk systems or where the developers and analysts have no prior experience. The first step in the process of Formal Methods is to develop Requirements State Machines or State Transition Diagrams.

A broad range of subtasks comprises Formal Methods. Those subtasks performed during software requirements development phase include the following:

Finite state machine/State Transition charts

Transaction Analysis

Proofs of Correctness

An introduction to Formal Methods is provided as Appendix-2 of the guidebook Detailed descriptions of Formal methods and state machines are given in the NASA Formal Methods Guidebook ."

4 3 Architectural Design Phase

The main safety objective of architectural design phase is to define the strategy for achieving the required level of fault tolerance in the different parts of the system. The degree of fault tolerance required can be inversely related to the degree of fault reduction used, e.g , Formal Methods. However, even the most rigorous level of fault reduction will not prevent all faults, and some degree of fault tolerance is generally required.

Independence / Failure Tolerance

NASA currently uses mostly hardware failure tolerance to control hazards The degree of hardware or system failure tolerance required varies with the severity of the hazard as follows.
Catastrophic Hazards. two- failure tolerance required
Critical Hazards single failure tolerance required.

These criteria are based on extensive experience of spacecraft flight operations which led to an accepted understanding of failure probabilities, and these levels of failure tolerance are accepted as necessary and sufficient to achieve an acceptable (low) level of risk

However, because of the unpredictable number of latent errors which might exists in software, software failure tolerance cannot be relied upon or verified in the same way Different hazard control approaches must be used for software versus hardware

To prevent fault propagation from uncontrolled software, SSCSCs must be fully independent of non-safety critical components

One approach is to establish "Fault Containment Regions" (FCRs) to prevent propagation of software faults. This attempts to prevent fault propagation such as· from non-critical software to SCCSCs; from one redundant software unit to another, or from one SCCSC to another Techniques known as "firewalling" should be used to provide sufficient isolation of FCRs to prevent hazardous fault propagation.

Methods of achieving independence are discussed in more detail in Reference [1] "The Computer Control of Hazardous Payloads", NASA/JSC/FDSD, 24 July 1991. FCRs are defined in reference [2]2 SSP 50038 Computer Based Control System Safety Requirements - International Space Station Alpha

[11] Gowen, Lon D. and Collofello, James S. "Design Phase Considerations for Safety-Critical Software Systems". PROFESSIONAL SAFETY, April 1995.

Gowen and Colldfield Reference [11] recommend four techniques for achieving fault-tolerance. Their paper is summarized below with permission, because it contains an excellent survey of the state of the art in these key areas.

Their five recommended techniques are·

N-Version programming
Recovery blocks
Resourcefulness
Abbott-Neuman Components.
Self-Checks

In addition, a summary of fault-tolerant solutions is given in Table 5.3.1 taken from reference [11].

4.3.1 1 N-Version Programming

This technique uses multiple software versions to tolerate runtime faults

N-Version programming is time consuming and expensive, as is maintaining multiple versions In addition, the different versions are not necessarily independent in their failures because different programmers tend to make similar errors, especially when errors are due to a flaw in the requirement's definition (Knight and Leveson [13], Brilliant, Knight and Leveson [14,15]) Under such conditions, the majority vote may be wrong, thus causing a hazard.

Despite its negative aspects, N-Version programming is useful for fault tolerance.

4.3.1.2 Recovery blocks

Like N-version programming, this technique uses multiple software versions to find and recover from faults In contrast, recovery blocks use an (internal) acceptance test on each version's output until output passes a test. The (internal) acceptance test uses reverse engineering to determine whether output is acceptable.

4.3 1.3 Resourcefulness

Resourcefulness concentrates on achieving system goals and requires systems that are functionally rich [21] Such a system can obtain its goals through multiple means For example, an airplane can descend by using its flaps to increase drag or decreasing its speed to reduce lift Resourcefulness is a system's ability to achieve goals via various known means so that the system can handle failures by trying different sub-goals

4 3.1 4 Abbott-Neuman Components

This technique combines various ideas: Abbott's software-component concept, Neumann's design criteria and software self-checks (Abbott [21], Neumann [22], Anderson [20], and Lee [TBD]) Abbott suggested that software focus can be the component level (i.e , module, package, task, etc.). which is where complexity originates. To increase a component's fault tolerance, Abbott applied Neumann's design criteria, which states that each component must be self-protecting and self checking A self protecting component does not allow other components to crash it, rather it returns an error indication to the calling component A self-checking component detects its own errors and attempts to recover from them.

4 3 1 5 Self Checks

Self-checks are not a fault-tolerant technique, but a classification of dynamic fault-detection categories, which various fault-tolerant techniques use. For example, N-version programming uses a replicative self-check, while recovery blocks use replication and either a reversal or reasonableness self-check.

Structural self-check is one that requires more explanation, it uses redundant data and checks to ensure that components manipulate complex data structures correctly.

(This concludes the summary of [11] Gowen, Lon D. and Collofello, James S "Design Phase Considerations for Safety-Critical Software Systems". PROFESSIONAL SAFETY, April 1995.)

4.4 Detailed Design Phase

The following tasks during the detailed design phases should support software safety activities.

1. Program Set Architecture.
2. Internal Program Set Interfaces.
3. Shared Data.
4. Functional Allocation
5. Error Detection and Recovery.
6. Inherited or Reused Software and COTS.
7. Design Feasibility, Performance, and Margins.
8. Integration
9. Interface Design.
10. Formal Methods - Formal specification Development (see 4 2 3)

4.5 Software Implementation

It is during software coding that software controls of safety hazards are actually implemented. Programmers must recognize not only the explicit safety-related design elements but should also be cognizant of the types of errors which can be introduced into non-safety-critical code which can compromise safety controls

4.6 Software Integration and Test

The safety testing effort should be limited to those software requirements classed as safety-critical items. Safety testing can be performed as an independent series of tests or as an integral part of the developer's test effort.

4.6 1 Testing Techniques

Testing should be performed in a controlled environment in which execution is controlled and monitored or in a demonstration environment where the software is exercised without interference.

4 7 Software Acceptance and Delivery Phase

Once the software has completed its acceptance testing it can be released either as a stand-alone item, or as part of a larger system acceptance.

Accompanying release of the software should be an Acceptance Data Package (ADP) This package as a minimum should contain a user manual.

5. Software Safety Analysis

During the software lifecycle, the software safety organization performs various analysis tasks, employing a variety of techniques. This section describes techniques which have been useful in NASA activities and some from elsewhere Some discussion on the cost and value of each technique is provided.

As software controls become more defined software hazard analyses will identify individual program sets, modules, units, etc which are safety-critical

5 1 Analysis of the Software Requirements for Potential Hazards

The requirements analysis activity clarifies and codifies safety requirements for the software and makes them consistent and complete

5 1.1 Software Safety Requirements Flowdown Analysis

5.1 2 Requirements Criticality Analysis

5.1 3 Formal Specification Analysis

Specification analysis evaluates the completeness, correctness, consistency, and testability of software requirements. Techniques used to perform specification analysis are:

hierarchy analysis,
control-flow analysis,
information-flow analysis, and
functional simulation

For the latter three techniques a large, well established body of literature exists describing in detail these methods, and many others, and background for each Instead of reproducing those lengthy texts the reader is directed to these excellent references·

Beizer, Boris, "Software Testing Techniques", Van Nostrand Reinhold, 1990. - (Note: Despite its title, the book mostly addresses analysis techniques).
Beizer, Boris, "Software System Testing and Quality Assurance", Van Nostrand Reinhold, 1987. (Also includes many analysis techniques).
Yourdon Inc., "Yourdon Systems Method - model driven systems development", Yourdon Press, N J., 1993.
DeMarco, Tom, "Software State of the Art. selected papers', Dorset House, NY, 1990.

5.1 4 Formal Inspections of Specifications

Formal inspections and formal analysis are different. Formal inspections are otherwise known as Fagan Inspections, named after John Fagan of IBM who devised the method NASA has published a standard and guidebook for implementing the Formal Inspection (FI) Process, Software Formal Inspections Standard (NASA-STD-2202-93) and Software Formal Inspections Guidebook (NASA-GB-A302) . FIs can and should be performed within every major step in the software development process However, they have the most value during the earlier requirements development phases, and decreasingly less value in later design and coding phases.

5.2 ARCHITECTURAL DESIGN ANALYSIS

The software architectural design process develops the high level design that will implement the software requirements

After allocation of the software safety requirements to the software design, Safety Critical Computer Software Components (SCCSCs) are identified

Analyses described for Architectural Design Phase are as follows

Update Criticality Analysis
Conduct Hazard Risk Assessment
Analyze Architectural Design
Interdependence Analysis
Independence Analysis

5.3 Detailed Design Analysis

During Detailed Design phase more detailed software artifacts are available, permitting rigorous analyses to be performed. Detailed Design Analyses can make use of artifacts such as the following· detailed design specifications, Pseudo-Code, emulators and

Program Description Language products (PDL). Preliminary code produced by code generators within case tools should be evaluated.

Many analysis techniques to be used on the final code can be "dry run" on these design products. In fact, it is recommended that all analyses planned on the final code should undergo their first iteration on the code-like products of the detailed design. This will catch many errors before they reach the final code where they are more expensive to correct. The following techniques can be used during this design phase.

Design Logic analysis
Software Fault Tree Analysis
Petri Nets
Dynamic Flowgraph Analysis
Markov Modeling
Design Data Analysis
Design Interface analysis
Measurement of Complexity
Design Constraint Analysis
Safe Subsets of Programming Languages
Formal Methods
Requirements State Machines
Formal Inspections

5.4 CODE ANALYSIS

Code analysis verifies that the coded program correctly implements the verified design and does not violate safety requirements In addition at this phase of the development effort, many unknown questions can be answered for the first time. For example, the number of lines of code, memory resources and CPU loads can be seen and measured, where previously they were only predicted, often with a low confidence level Sometimes significant redesign is required based on the parameters of the actual code

Code permits real measurements of size, complexity and resource usage.

Some of the techniques used in the performance of code analysis mirror those used in design analysis However, the results of the analysis techniques might be significantly different than during earlier development phases, because the final code may differ substantially from what was expected or predicted.

1 DESIGN LOGIC ANALYSIS
2 Software Fault Tree Analysis (SFTA)
3 Petri-Nets
4 Design Data Analysis
5 Design Interface Analysis
6 Measurement of Complexity

7 Design Constraint Analysis
8 Safe Subsets of Programming languages
9 Formal Methods and Safety-Critical Considerations
10 Requirements State Machines

Each of the analyses in this section should be undergoing their second iteration, since they should have all been applied previously to the code-like products (PDL) of the detailed design.

5.5 TEST ANALYSIS

Two sets of analyses should be performed during the testing phase.

1) analyses before the fact to assure validity of tests and, 2) analyses of the test results

5.6 SOFTWARE OPERATIONS & MAINTENANCE

Maintenance of software differs completely from hardware maintenance Unlike hardware, software does not degrade or wear out over time, so the reasons for software maintenance are different.

The main purposes for software maintenance are as follows:

to correct known defects
to correct defects discovered during operation
to add or remove features and capabilities (as requested by customer, user or operator)
to compensate or adapt for hardware changes, wear out or failures

The most common safety problem during this phase is lack of configuration control, resulting in undocumented and poorly understood code. "Patching" is a common improper method used to "fix" software "on the fly". Software with multiple undocumented patches has resulted in major problems where it has become completely impossible to understand how the software really functions, and how it responds to its inputs.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 1995

3 REPORT TYPE AND DATES COVERED
Final Contractor Report

4. TITLE AND SUBTITLE
Software Safety Progress in NASA

5. FUNDING NUMBERS
WU-601-60-60
C-NAS3-26764

6. AUTHOR(S)
Charles F. Radley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Raytheon Engineers and Constructors
2001 Aerospace Parkway
Brook Park, Ohio 44142

8. PERFORMING ORGANIZATION REPORT NUMBER
E-9899

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-198412

11. SUPPLEMENTARY NOTES
Prepared for the Safety Through Quality Conference cosponsored by NASA, BCS, and RTAL, Cape Canaveral, Florida, October 23–25, 1995 and the Workshop on Safety and Reliability in Emerging Control Technologies, sponsored by the International Federation of Automatic Control, Dayton Beach, Florida, November 1–3, 1995. Project Manager, Martha Wetherholt, Assurance Engineering Office, NASA Lewis Research Center, organization code 0520, (216) 433–2416

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Categories 01, 03, 12, 17, 18, 31, 38, 61, 62, and 66
This publication is available from the NASA Center for Aerospace Information, (301) 621–0390

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

NASA has developed guidelines for development and analysis of safety-critical software. These guidelines have been documented in a Guidebook for Safety Critical Software—Development and Analysis The guidelines represent a practical "how to" approach, to assist software developers and safety analysts in cost effective methods for software safety. They provide guidance in the implementation of the recent NASA Software Safety Standard NSS–1740.13 which was released as "Interim" version in June 1994, scheduled for formal adoption late 1995. This paper is a survey of the methods in general use, resulting in the NASA guidelines for safety critical software development and analysis

14. SUBJECT TERMS
Software; Safety; Assurance; Analysis; Verification, Systems; Development; Critical

15 NUMBER OF PAGES
20

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

End of Document