# Distribution Of Defects In A Large Software System

**Stephen Wickham**

B.Sc, University of Victoria, 1997

Project Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master Of Science

in

Mathematics, Computer And Physical Sciences

(Computer Science)

UNIVERSITY of NORTHERN
BRITISH COLUMBIA
LIBRARY
Prince George, B.C.

The University Of Northern British Columbia

January 2007

© Stephen Wickham, 2007

Abstract

This report summarizes the findings of a retrospective analysis of coding errors in a major software system produced by a large Canadian software engineering firm. The code-base of the system is approximately 1.7 million lines of C++ integrated with third party RDBMS and GIS products. The safety related nature of the system and the size of its code base make it an ideal candidate for an investigation of software related defects. The investigation focuses primarily on memory management related defects referred to as "memory leaks." A "memory leak" results from the failure to return previously allocated heap memory. The distribution of memory leaks is analyzed and a two-part memory leak classification scheme is described. A secondary focus of the investigation is the influence of decision complexity on system safety. This investigation yielded two statistically significant findings. The first is a relationship between programmer experience and memory leak creation. The second is a correlation between subsystem complexity and memory leak density. The impact of software process improvement measures are also discussed.

This document contains information proprietary to MacDONALD, DETTWILER AND ASSOCIATES LTD., to its subsidiaries or to a third party to whom MacDONALD, DETTWILER AND ASSOCIATES LTD. may have a legal obligation to protect such information from unauthorized disclosure, transfer, export, use or duplication. Any disclosure, use or duplication of this document, or any of the information contained herein, for other than the specific purpose for which it was disclosed is expressly prohibited, except as MacDONALD, DETTWILER AND ASSOCIATES LTD. may otherwise agree to in writing.

TABLE OF CONTENTS


**I. Introduction ........ 1**

1.1 Motivations ........ 1

1.2 Document Organization ........ 2

**II. Background and Literature Review ........ 3**

2.1 System Overview ........ 3

2.2 Historical Context ........ 5

2.3 Debug & Release Builds ........ 5

2.4 Memory Management ........ 7

2.4.1 Static Memory ........ 7

2.4.2 Automatic Memory ........ 7

2.4.3 Free Store ........ 7

2.5 Software Design Patterns ........ 9

2.6 Software Safety and Reliability ........ 10

2.6.1 Definitions ........ 11

2.6.2 Complexity & Safety ........ 12

2.6.3 Modularity & Safety ........ 18

2.6.4 Compiler Standards Compliance ........ 19

2.7 Software Testing ........ 20

2.7.1 Definitions ........ 20

2.7.2 Testing Strategies ........ 22

2.7.3 Static Analysis ........ 22

2.7.4 Dynamic Analysis ........ 23

2.8 Software Process ........ 24

2.8.1 Process Models ........ 24

2.8.2 Process Measurement ........ 25

2.8.3 Process Improvement ........ 26
2.8.4 Process Engineering ........ 28
**III. Methodology ........ 30**
3.1 Memory Leak Detection ........ 30
3.1.1 Dynamic Analysis ........ 30
3.1.2 Static Analysis ........ 32
3.2 Memory Leak Classification ........ 34
3.3 ANSI C++ Compliance ........ 36
3.4 Metrics ........ 37
3.4.1 Memory Leak Density ........ 37
3.4.2 Complexity Measurement ........ 37
3.4.3 Programmer Experience ........ 38
**IV. Experiments and Results ........ 39**
4.1 Memory Leak Distribution ........ 39
4.2 Memory Leak Density & Subsystem Complexity ........ 42
4.2.1 Dataset ........ 42
4.2.2 Experimental Details and Results ........ 44
4.3 Defect Density & Subsystem Complexity ........ 46
4.4 Memory Leaks & Programmer Experience ........ 47
4.4.1 Dataset ........ 47
4.4.2 Experimental Details and Results ........ 47
4.5 Process Improvement & Defect Density ........ 49
4.6 ANSI C++ Compliance ........ 51
**V. Conclusions ........ 53**
5.1 Contributions ........ 53
5.2 Findings ........ 53
5.3 Future Work ........ 55
**VI. Bibliography ........ 56**

VII. **Appendix 1: Non Disclosure Agreement ..................................................... 61**

## LIST OF TABLES

1. FAA failure probability values .......... 13

2. Cyclomatic Complexity Values .......... 16

3. Halstead Complexity Measures .......... 17

4. Conte's Modularization Levels .......... 19

5. SEI CMM Levels .......... 25

6. Review Process Description .......... 27

7. Life-cycle Leak Classification Categories .......... 35

8. Dataset field descriptions .......... 42

9. Complexity Density versus Leak Density Correlation Data .......... 43

10. Chi Squared Analysis of Programmer Experience .......... 48

11. Release B vs. C Defect Densities .......... 49

12. Software defects identified by compiler error .......... 51

13. ANSI Compliance Error Examples .......... 52

LIST OF FIGURES

1. System Architecture Diagram ........ 4

2. Memory Layout ........ 8

3. FAA failure probability chart ........ 13

4. Example control graph ........ 16

5. Review Process Flowchart ........ 28

6. Memory Leak Distribution ........ 40

7. Leak Density vs Complexity Density Scatter Plot ........ 44

8. Defect Density vs Subsystem Complexity ........ 46

# ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Siamak Rezaei, for his guidance, encouragement, and support throughout the course of this research. In addition I would like to thank Dr. Robert Tait for his careful reviews and constructive feedback. I would also like to acknowledge MacDonald Dettwiler & Associates (MDA) for granting access to the intellectual property upon which this investigation is based. Finally, I would like to thank Mr. Michael Lingren for sharing his hard won insight and wisdom gained in the real world of software engineering.

# GLOSSARY

| | |
|---|---|
| **RDBMS** | Relational Database Management System |
| **GIS** | Geographic Information System |
| **KSLOC** | Thousand Source Lines Of Code |
| **SEI** | Software Engineering Institute |
| **SLOC** | Source Line Of Code |
| **RFP** | Request For Proposal |

# Chapter 1

# Introduction

## 1.1 Motivations

Memory management defects are a significant impediment to the successful creation of any release build and to reliable software in general [MS06]. As such, this investigation placed major emphasis on coding practices that introduced memory management problems. Findings made during this process are discussed in the context of software safety, reliability, and correctness. The distribution of errors in the code base is compared with developer experience and code complexity. This paper argues that memory management related defects are related to code complexity and programmer experience.

Approximately 31% of software projects will be cancelled prior to completion [Sta94]. Given this, it is easy to see why a mature, fielded, software system that has undergone structured verification and customer acceptance is extremely valuable. In the academic context the code base of such a system represents an ideal research dataset. The findings resulting from this investigation are all the more relevant because they are derived from a real world system. In spite of the latent defects identified by this research, we must recognize all that has been done correctly to get such a large system out the door and into the hand of the customer. The findings presented here can teach us as much about what has worked well in the past as they can about what can be improved in the future.

Software permeates almost every aspect of our daily lives. From the microprocessor in a car to the instrument approach landing system that guides a flight safely to the ground, our lives depend on code. To those industry and academic professionals involved in the field, this is can be a disquieting thought indeed. With software, come defects. That is the unavoidable reality of the discipline. Some very notable accidents, some of which unfortunately involve the loss of human life, have been traced back to software defects. Consider the following: In June 1996, a Washington DC subway driver was killed when his automatically controlled train slammed into a wall instead of stopping at the last station [Her97]; On November 24th 1991, a British newspaper reported that radiation safety doors at the Sellafield nuclear facility had been opened accidentally due to a computer error [Hat95]. In short, software defects are a very real problem.

## 1.2 Document Organization

The following section describes the overall structure of this document. The paper is composed of five chapters and one appendix that are described below.

Chapter 1 presents introductory material, describes project motivations, and outlines overall document structure.

Chapter 2 discusses background topics relevant to this project including software safety and reliability, static analysis, dynamic analysis, complexity measurement, memory management, and software defect classification.

Chapter 3 describes the methodology employed during this investigation. The memory leak detection framework used is described along with static analysis tools and techniques. Problems and challenges associated with each of the techniques are also discussed.

Chapter 4 presents findings and discusses results. Error distribution is discussed and correlated with developer experience and code complexity.

Chapter 5 concludes the investigation with recommendations for future software engineering projects and suggestions for future work.

Appendix 1 contains the non-disclosure agreement between MDA and UNBC that governs the dissemination of intellectual property contained herein.

# Chapter 2

# Background and Literature Review

## 2.1 System Overview

This paper documents the retrospective analysis of a large Windows based software product. The product, herein referred to as "the system", aids in the design of instrument approach procedures for the aviation industry. The code-base is approximately 1.7 million lines of C++ integrated with third party RDBMS and GIS products. Object-oriented design principles are employed throughout. The system is implemented as a series of subprojects, each represented by a dynamic link library. The subprojects are grouped into layers as follows:

- Data Management
- GIS and Geometry
- Common Aeronautics
- Procedure and Chart Design
- User Interface

The dataset used in this analysis was derived using the system's automated regression test facility. Because the test facility does not exercise the user interface, this layer was excluded from the analysis. Core system functionality in all the remaining layers is included. The system architecture diagram below depicts high-level structure and component relationships.

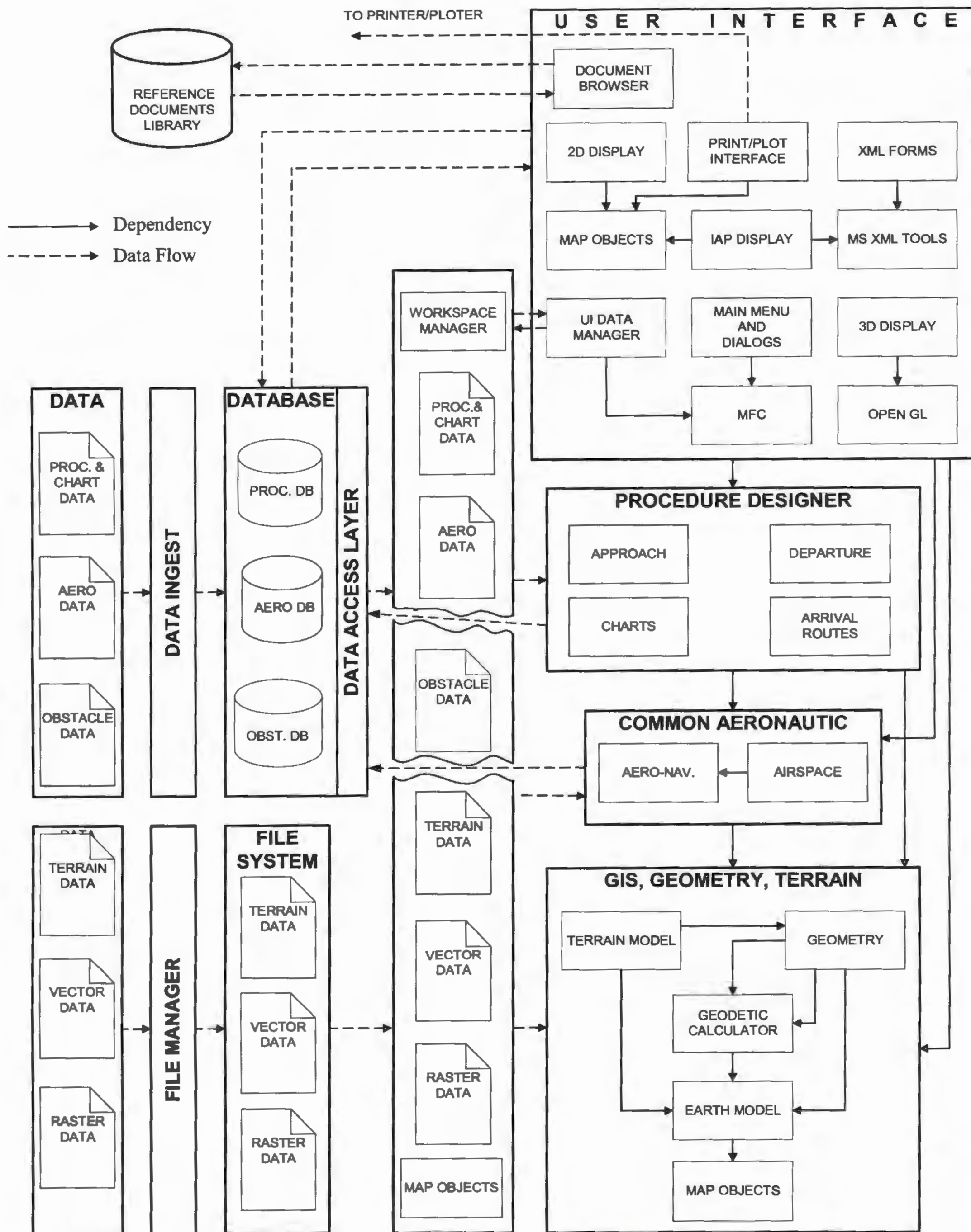


Figure 2-1: System Architecture Diagram

The data access and file systems layers provide an abstraction barrier between the physical storage of data and its representation within the system. The GIS layer interacts with the file system to retrieve digital terrain data and provides numerical geometry services utilized by higher level subsystems. The Aeronautical and Procedure Design layers implement core system algorithms, supported by the database, file system, and GIS layers. The User interface layer sits at the highest level and interacts with lower level subsystems as needed. The Data Ingest subsystem manages external system interfaces.

## 2.2 Historical Context

The system has been in development for five years and has undergone three major releases. Due to schedule and budget constraints a release build of the system was not produced during the initial stages of development. By the time an attempt was made to produce a release build the size of the code base had grown significantly. Programming practices that produced code incompatible with a release build were well entrenched. The severity of problems encountered and volume of offending code caused the original release build to be abandoned. Data used in this investigation was collected during a subsequent release build attempt.

## 2.3 Debug & Release Builds

Generally accepted software engineering practice dictates that source code is compiled or "built" in two separate configurations: Debug & Release [Pet99]. A "Debug" build is instrumented with symbolic debugging information that makes diagnosis of software defects considerably easier. Symbols facilitate interactive debugging by providing source code, line number, variable, and data type information during program execution [ASU88]. In the Microsoft environment memory management is tracked using a debug heap that buffers memory allocations, tracks memory leaks, and protects the programmer from access

violations, etc. This instrumentation imposes significant performance overhead and requires the distribution of supporting debug libraries.

Debug builds also disable compiler optimizations. Compiler optimizations are "improvements" introduced by the compiler in order to increase the speed or reduce the size of generated object code [ASU88]. Optimizations that are disabled in a debug build but enabled in a release build can be responsible for altered behaviour between the two configurations.

In contrast, a "Release" build is un-instrumented and is optimized for either size or speed. Generally speaking a release build will run considerably faster than a corresponding debug build. It will have a smaller memory footprint, smaller executable size, and will not require the distribution of supporting debug libraries.

By their very nature, debug builds are more tolerant of poor programming practice. Code that compiles and runs (although perhaps not correctly) in a debug build may crash in a release build. In general, release builds and debug builds are developed in parallel from project inception. In this way problems that prevent a release build from compiling or running can be dealt with as they arise.

## 2.4 Memory Management

C++ defines three memory management mechanisms: static memory, automatic memory, and the free store [Str97]. These mechanisms and their relevance to this project are described below.

### 2.4.1 Static Memory

Items allocated in static memory persist for the duration of program execution. Static class members, static variables in functions, and global variables all reside in static memory. Static memory is relevant to this investigation as it relates to singleton allocations (see section 2.5).

### 2.4.2 Automatic Memory

Automatic or "stack allocated" memory is used to store local variables and function arguments. Items allocated on the stack are automatically created and destroyed as they come in and out of scope. Automatic memory is safe, simple, and faster than heap allocation [MR94]. In fact, 10% of the leaks discovered during this investigation involved heap allocations that could have been made on the stack.

### 2.4.3 Free Store

The free store or "heap" is an area of memory used for dynamic allocation. Heap allocation is used when the number and size of blocks in not known until runtime. Allocations and de-allocations are made explicitly using the new and delete operator respectively. The free store is finite and is ultimately limited by the resources of the host system. Once the program has finished using memory allocated on the free store it must give that memory back. Failure to do will cause problems, particularly in long running programs. The failure to return previously

allocated heap memory is referred to as a "memory leak". The analysis of programming practices leading to memory leaks, and the correlation of memory leaks with code complexity is the major focus of this paper.

Figure 2-2 below illustrates typical memory layout. Memory is partitioned into address ranges dedicated to each type of storage. Program code and static variables are grouped together in static memory. A separate region, referenced by the stack pointer, is dedicated to the program stack and associated "automatic" memory. Finally, the "heap" or free store contains dynamically allocated memory referenced via pointers. In the example below, two dynamic memory allocations are referenced via static pointer variables.

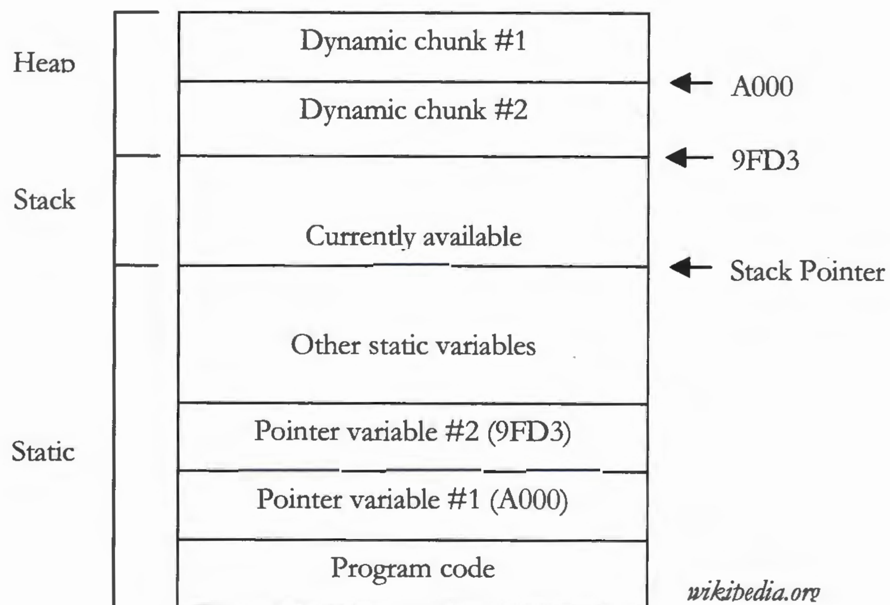


Figure 2-2: Memory Layout

Memory allocated on the heap has two interesting properties: it is unnamed and un-initialized [Lip92]. In this context the "unnamed" property means the memory must be manipulated indirectly via pointers. The "un-initialized" property means just that, the memory is not set to any predefined value upon initialization. Together these properties are responsible for a great deal of unintended system behaviour [Koe88][Hat95][Mco93]. Many debug libraries attempt to protect the programmer from poor pointer management by creating buffer zones and filling newly allocated memory with a predefined dummy values. Strategies such as this are helpful for diagnosing memory management problems during system development but can foster a false sense of security. Once the protection provided by the debug library is removed, a system that appeared stable in debug configuration can become unusable.

## 2.5 Software Design Patterns

The concept of a design pattern has its basis in building architecture and was originally conceived of by Christopher Alexander. Alexander defines a design pattern as in the following manner, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, with ever doing it the same way twice" [Ale77].

Essentially, a design pattern captures the essence of a problem's solution in an abstract and re-usable way. Just as component based development and object oriented design foster code reuse, design patterns foster design re-use. While design patterns are not a silver bullet, appropriate pattern application can make a significant contribution to software quality [Ris98][Lar02]. Specifically, pattern usage promotes abstraction, code re-use, and ease of maintenance.

The original software engineering design patterns catalogue was published in 1995. Its authors, Gamma, Helm, Johnson, and Vlissides are commonly referred to as the Gang of Four (GoF). Their book publishes 23 patterns that have been the focus of countless academic endeavours and are ubiquitous in the world of software engineering. The system studied here utilizes a number of the creational, structural, and behavioural patterns in this catalogue. The "Singleton" pattern discussed below is relevant to the discussion of memory management.

The singleton design pattern ensures a class has only one instance, and provides a global point of access to it [GHJ+95]. This pattern is used widely throughout the system in question. The implementation of the singleton pattern studied here uses a static pointer to a heap allocated object. This implementation created problems as the static pointer was never explicitly destroyed and thus, the associated heap allocation was never freed. In practice, this situation is benign, as a leak that occurs at program termination cannot impact program execution in any meaningful way. It did, however, create spurious errors during dynamic analysis as benign singleton leaks were reported. Management of singleton lifetime is a common problem in pattern based development [Ken03]. Numerous solutions have been presented, the most notable being Alexandrescu's Loki design [Ale01].

## 2.6 Software Safety and Reliability

Before discussing formal definitions of software safety some historical background on software related accidents might help to provide context:

- A defect in the control software of the Therac-25 radiation therapy machine resulted in a number of patient deaths. The defect was ultimately traced to a race condition [Lev93].

- An Ariane 5 rocket exploded 37 seconds after launch due to an unhandled exception when a 64 bit floating point variable was assigned to a 16 bit unsigned integer [ESA96].

These accidents highlight the catastrophic effects that software defects can have. Indeed it has been suggested that, under certain circumstances, the safest thing to do may be avoid software all together [Hat95].

### 2.6.1 Definitions

Software safety has been defined in numerous ways by numerous authors. Herrmann defines software safety as follows [Her99]:

> *"features and procedures which ensure that a product performs predictably under normal and abnormal conditions, and ensure the likelihood of an unplanned event occurring is minimized and its consequences controlled and contained thereby preventing accidental injury or death, whether intentional or unintentional"*

The use of the term 'minimized' in the definition above is interesting. It hints at the difficulty in achieving absolute reliability. This subject is discussed further below.

Hatton defers to the Concise Oxford Dictionary:

> *reliability:* 'Of sound and consistent character or quality'
>
> *safety:* 'Freedom from danger or risks'

Hatton is careful to draw a distinction between reliability and safety. He states a program can be unreliable but safe, meaning its defects do not create risk; or a program can be reliable but unsafe, meaning it will produce consistently incorrect and potentially harmful results.

Conte distinguishes the terms fault and defect as follows [Con86]:

*fault:* 'an error that causes an incorrect result for a valid input'

*defect:* 'evidence of the existence of a fault'

These definitions suggest that faults may be latent, but that defects do not exist until they actually manifest themselves in some measurable way.

### 2.6.2 Complexity & Safety

Software safety and reliability problems have a wide range of causes. Lack of process and structure, poorly defined requirements, and semantic idiosyncrasies of the implementation language can all be contributing factors. An in depth discussion of safety and reliability is beyond the scope of this work. One topic, however, is of particularly relevance to this investigation: the influence of complexity on safety.

The complexity of many safety critical systems (e.g. a nuclear reactor control system) is so high that it is simply not possible to guarantee that software is completely error free [LS93]. As a result, software specifications typically place an upper bound on the probability of failure. The following excerpt from FAA Advisory Circular 25.1309-1A on Systems Analysis and Design illustrates the probabilistic nature of reliability specification [FAA88]:

(1) Minor failure conditions may be probable.

(2) Major failure conditions must be improbable.

(3) Catastrophic failure conditions must be extremely improbable.

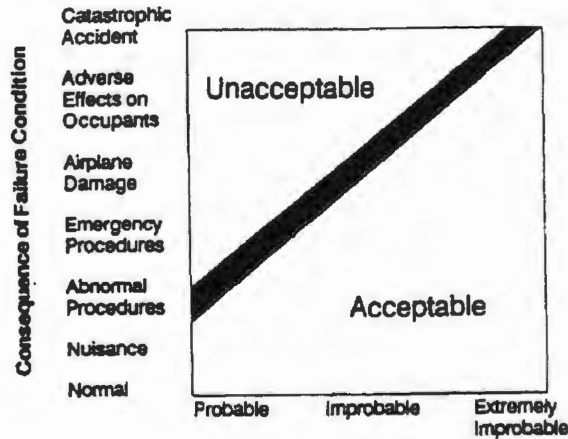


Figure 2-3: FAA failure probability chart

This specification assigns the following definitions and associated probabilities:

Table 2-1 FAA failure probability values

| Type | Description | Probability |
|---|---|---|
| Probable | Anticipated to occur one or more times during the entire operational life of an aircraft | $> 1 \times 10^{-5}$ |
| Improbable | Not anticipated to occur during the lifetime of a single random airplane, but anticipated to occur during the entire operational lifetime of all aircraft of one type | $\leq 1 \times 10^{-5}$<br>$> 1 \times 10^{-9}$ |
| Extremely Improbable | Not anticipated to occur during the entire operational lifetime of all aircraft of one type | $\leq 1 \times 10^{-9}$ |

The negative impact of complexity of software safety is a recurring theme in software engineering literature.

Conte defines five design principles, one of which is complexity. He states: "*A design should be kept as simple as possible. Design complexity grows as the number of control constructs grows. The hypothesis is that designs with high complexity will contain more errors*" [Con86].

Leveson re-iterates the importance of keeping designs simple in her commentary on the Therac-25 accidents described above. She states: "*Basic software-engineering principles that apparently were violated with the Therac-25 include: ...Designs should be kept simple*" [Lev93].

Hatton defines the terms $C_n$, the *natural complexity* inherent in a problem, and $C_a$, the *actual complexity* used solve it. While the inequality $C_a \geq C_n$ will always hold, he asserts that the difference between $C_a$ and $C_n$ should be kept as small as possible [Hat95]. In plain English this means the complexity of software used to solve a problem should be kept as close as possible to the inherent complexity of the problem itself.

In addition to the discussion above, Banker et al. provide an excellent survey of software maintenance related research [BDK+02]. Their meta-analysis cites numerous publications that report a positive correlation between software complexity and elevated defect rates.

Gitten et al. conducted experiments similar to those described in Chapter 4. Their experiments attempted to establish a correlation between general defect densities (e.g. not specific to memory leaks) and code complexity. Interestingly, they were not able to find an observable relationship between the two parameters [GKG05]. They did, however, establish that the Pareto Principle (80:20 rule)

appears to apply to software defect distribution. Specifically, they found 100% of the defects in 28% of the code, and 80% of the defects in 26% of the code. Ostrand et al. also failed to find a correlation between code complexity and defect rates [OWB04].

#### 2.6.2.1 Complexity Metrics

The metrics of software complexity receive thorough treatment in the software engineering literature. Complexity metrics exists to measure both logical and computational complexity.

##### 2.6.2.1.1 Logical Complexity Metrics

Logical complexity metrics quantify the complexity of a program's decision structure. Hatton distils the long list of available metrics to three: cyclomatic complexity, static path count, and fan-in/fan-out [Hat93]. These are discussed below.

###### 2.6.2.1.1.1 Cyclomatic Complexity

McCabe originally described cyclomatic complexity in his classic 1976 paper entitled "A Complexity Measure" [Mca76]. Cyclomatic complexity uses graph theory to describe a program's decision complexity. The commands of the program are represented as nodes in the graph. If one command may execute a second command, the two nodes representing the commands are connected by an edge in the graph. The formal definition of cyclomatic complexity is as follows [SEI06]:

Cyclomatic Complexity = E - N + 2; where

E = the number of edges in the graph

N = the number of nodes in the graph

For example, an if/else statement with the following control graph would have a cyclomatic complexity value of 2 (4 edges – 4 nodes + 2):

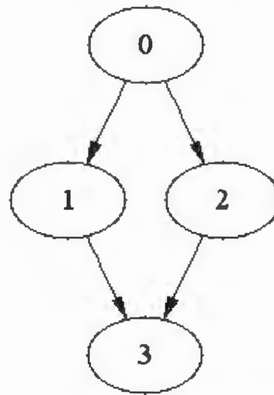


Figure 2-4: Example control graph

The Carnegie Mellon Software Engineering Institute provides the following guidelines for interpreting cyclomatic complexity values [SEI06]:

Table 2-2 Cyclomatic Complexity Values

| Cyclomatic Complexity | Risk Evaluation |
|---|---|
| 1-10 | a simple program without much risk |
| 11-20 | more complex, moderate risk |
| 21-50 | complex, high risk program |
| > 50 | untestable program (very high risk) |

Cyclomatic complexity is widely accepted as a predictor of defects and as a measure of the difficultly in maintaining code [BDK+93][KS97].

#### 2.6.2.1.1.2 Static Path Count

Static path count is simply the number of distinct paths through a program. The metric assumes all predicates are independent [Hat93].

#### 2.6.2.1.1.3 Fan-in/Fan-out

Fan-in/Fan-out measures the number of times a particular function is referenced, and the number of functions it references. Hatton defines the associated metric as fan-in + fan-out + (fan-in x fan-out) [Hat93].

### 2.6.2.1.2 Computional Complexity Metrics

Computational complexity metrics quantify a program's calculation complexity. This type of metric is most appropriate in cases where program code contains more calculation logic than branching logic.

The Halstead Complexity Measure is an excellent example of a computation complexity metric. This metric quantifies complexity based on operators and operands used in program source code. The metric is based on the following parameters [Hal77][SEI06]:

n1 = the number of distinct operators

n2 = the number of distinct operands

N1 = the total number of operators

N2 = the total number of operands

from which the following five measures are computed:

Table 2-3 Halstead Complexity Measures

| Measure | Symbol | Formula |
|---|---|---|
| Program Length | N | N = N1 + N2 |
| Program Vocabulary | N | n = n1 + n2 |
| Volume | V | $V = N \times (\log_2 n)$ |
| Difficulty | D | (n1/2) x (N2/n2) |
| Effort | E | D x V |

Halstead's complexity measures differ from McCabe's cyclomatic complexity in terms of their suitability. Halstead metrics focus on operators and operands. As such they are better suited to measuring computational complexity. McCabe metrics on the other hand are more focused on logical complexity [SEI06].

### 2.6.3 Modularity & Safety

The conventional wisdom in software engineering is that modularity is good [Mco93]. However, both Hatton and Conte present evidence that suggests over modularization can be as harmful as under modularization [Con86][Hat93]. Hatton asserts that proportionally more errors are committed in small software components than large ones. He defines the following logarithmic relationship between the number of static paths and the number of software defects in a module where $n_b$ is the number of bugs, $n_p$ is the number of static paths, and $c$ is a constant close to 1:

$$n_b = c \log_{10}(n_p)$$

This formula predicts that a single module with a complexity of 100 will have less defects that 10 modules each with a complexity of 10. This relationship is significant as it describes the interaction between modularity, complexity, and defect count.

Conte defines three levels of modularization:

Table 2-4 Conte's Modularization Levels

| Modularization Level | Description |
|---|---|
| Unmodularized | the entire program is written as one routine |
| Partially modularized | the program is broken up into a "moderate" number of subroutines |
| Super modularized | the program is broken up into twice the number of subroutines as the partially modularized version |

He has illustrated that the best level of reader comprehension is achieved with a partially modularized version of a program [Con86]. This is relevant as comprehension is a necessary prerequisite to successful program maintenance. These somewhat counter-intuitive findings illustrate that modularity plays an importance role in software safety, but that modularity is perhaps best applied in moderation.

### 2.6.4 Compiler Standards Compliance

The extent to which a compiler adheres to a language standard is a key determinant of software safety [Hat95]. The Microsoft Visual C++ 6.0 (VC 6) compiler used for initial system development did not conform to the ANSI C++ standard in a number of significant ways [ANSI03]. Visual C++ 8.0 (VC 8) supersedes VC6 and exhibits improved ANSI compliance. As a result of this improved compliance, a number of breaking changes have been introduced. Code that compiled under the old compiler is now considered illegal. Almost without exception, code that no longer compiles with VC 8 is in some way incorrect. This finding is consistent with the assertion that the compiler itself is often the simplest and most effective debugger [Mco93][Mey98]. As such porting the code base from VC 6 to VC 8 made a major contribution to system correctness and achieving the goal of a stable release build. The nature and

distribution of breaking changes corrected during the compiler upgrade are discussed at length in Chapter 4.

## 2.7 Software Testing

On average, software testing consumes at least 50% of the effort required to produce a working, fielded software system [Bez90]. Given this, and given that this investigation is concerned with the distribution of software defects, some discussion of the topic is warranted. Having said this, software testing is a vast topic that can only be given superficial treatment here. Portions of the discipline that are of particular relevance to this investigation are discussed in this section.

### 2.7.1 Definitions

The terms "verification" and "validation" and "testing" are often used together in the context of software evaluation. Marks provides the following definitions of some commonly used (and confused) terms [Mar92]:

*validation:* 'A determination of the correctness of the final product produced by a development project with respect to the user's needs and requirements.'

*verification:* 'A demonstration of the consistency, the completeness, and the correctness of the system ...'

*testing:* 'An examination of the behaviour of a system by executing the system on a sample set'

The distinction between "verification" and "validation" is of particular interest. Verification involves checking that the software correctly implements its requirements. Validation involves ensuring that the software meets the customer's needs [Boe78][Som01]. A software product may dutifully implement a set of requirements and remain completely unusable. The concept of "fit for

purpose" is relevant here. This idea accepts that software will never be completely defect free, but states it must ultimately meet the needs of its users.

The word *"sample"* in the definition of *"testing"* above also warrants discussion. The extent to which a test samples the program code it is evaluating is referred to as the test's *code coverage*. Since it is impossible to completely test any reasonably complex software system, a subset of tests that has the highest probability of detecting the most errors should be identified [Kit95].

The concept of regression testing must be also introduced. Bezier defines regression testing as follows [Bez90]:

> *"any repetition of tests (usually after software or data change) intended to show that the software's behaviour is unchanged except insofar as required by the change to the software or data"*

A large portion of the dataset used in this investigation was collected using the system's automated regression testing capability.

Finally, some definitions of system size are relevant. The size of a system is a major factor in determining a test strategy. Marks provides the following definitions of system size in his book "Testing Very Large Systems" [Mar92]:

*small:* 'A small system provides a single service, such as an inventory system, an accounting system, or a billing system. Typically, a small system is under 500 000 lines of code.'

*medium:* 'A medium system is one that provides several services. The major difference between a small system and a medium system is the functionality and integration of the functionality into a user-oriented package. Typically, a medium system is 500 000 to 2 million lines of code.'

*large:* 'A big software system provides many services. The major difference between a medium and a big system is the complexity

> of the functions. Complexity can be caused by complex computational algorithms or by complex data relationships. Typically a big system is more than 3 million lines of code.'

On the basis of line count alone, the system studied here would be considered medium sized. However, the complexity of its algorithms, its safety related nature, and high customer expectations mean it exhibits many of the properties Marks attributes to large systems.

### 2.7.2 Testing Strategies

Two fundamental test strategies exist: "Black-box" testing and "White-box" testing. Black-box testing evaluates software against its requirements. Tests of this type are derived from the software's specification rather than its internal structure. Black-box tests are meant to take an objective, independent view of software from a requirements based perspective. As such they should be written in isolation from program code to the greatest extent possible [Kit95].

In contrast, White-box testing requires knowledge of internal program structure. Tests of this type are concerned with internal issues such as code coverage, logical incompleteness, decision complexity, etc. White-box testing permits the test author to choose input values that will most effectively exercise all braches of program logic [Kit95]. The static and dynamic analysis techniques discussed below also fall into this category of testing.

### 2.7.3 Static Analysis

Static analysis describes the semantic and syntactic inspection of program source code to detect software defects prior to program execution. The term "Automated Software Inspection (ASI)" is also used to refer to this approach. The first and most well known static analysis tool is Lint, created by Johnson at

Bell Labs in 1978 [Joh78]. Lint was designed to detect suspicious programming constructs and non-portable code. Since then, commercial static analysis tools have proliferated. These tools support a wide range of languages and many allow the user to define custom rules that are specific to the coding standards of the organization in question. In a sense, all code that is compiled undergoes rudimentary static analysis. Object code cannot be generated from syntactically illegal source, and can therefore never be executed. Some languages attempt to increase the number of potential defects that can be detected statically. Strongly typed languages such as Ada forbid implicit type conversions and make constructs such as incomplete switch statements illegal. In this way, language definition can make a significant contribution to software safety [Hat93]. All of the metrics described in section 2.7 above can be measured statically. Static analysis techniques have been shown to effectively predict failures and fault prone modules [NWH+04].

### 2.7.4 Dynamic Analysis

In contrast to static analysis, dynamic analysis describes the process of detecting faults during program execution. This is a far less desirable state of affairs, as the likelihood of detecting a defect at runtime depends on the quality of testing. By definition, a fault that is only detectable at runtime may not be detected at all [Hat93]. Many commercial tools exist to facilitate dynamic analysis. While this investigation is concerned with memory management related defects, dynamic analysis techniques can be applied to other functional areas including code coverage and performance analysis. Two of the best-known dynamic analysis tools for memory management are Rational's Purify and Parasoft's Insure++ tools. Most dynamic analysis tools work by instrumenting object code with constructs that allow the tool to track where and when a particular defect originates. In addition to the source code file and line number of the leak, the

most valuable data a dynamic analysis tool can report is the call stack associated with the leak. The call stack provides crucial information on state of program execution at the time the leak occurred. Interestingly, the Microsoft debug libraries utilized in this investigation do not work by instrumenting object code. Instead, the runtime library itself tracks the state of the heap. Unfortunately the Microsoft debug library reporting functions to not provide call stack information. As such all of the leak data used in this investigation was gathered without the benefit of a call stack. Dynamic analysis tools used for code profiling and performance tuning include Intel's vTune and Microsoft's Profile Guided Optimization.

## 2.8 Software Process

The Oxford Dictionary of English defines a process as a series of actions or steps taken in order to achieve a particular end. In the case of software engineering, the end is typically the creation of a software product. This section discusses software process topics relevant to this investigation.

### 2.8.1 Process Models

A software process model is an abstract representation of a software process [Som01]. Well known software process models include the waterfall model, the evolutionary (or spiral) model, re-use (or component) based development, and formal methods. Many large software projects take a blended approach, using different techniques for different parts of the system. The classic software process models mentioned above are well documented in standard software engineering textbooks and will not be discussed in detail. In this context it is sufficient to identify elements common to all software process models. To a greater or lesser extent, all models address the core topics of specification, design, implementation, and verification.

### 2.8.2 Process Measurement

On a large project organizational characteristics eclipse the attributes of any single individual. As such the quality of the software process governing the team becomes a large determinant of the project's success [Kit95][Eva04]. There is a growing body of work dedicated to the assessment and evaluation of software process. For example, the Carnegie Mellon Software Engineering Institute (SEI) introduced the concepts of organizational maturity and capability. These concepts and the associated five level Capability and Maturity Model (CMM) are used to quantify an organization's process model. The SEI's description of each level (paraphrased by [Kit95]) is outlined below:

Table 2-5 SEI CMM Levels

| Level | Description |
|---|---|
| 1: Initial | Unpredictable and poorly controlled |
| 2: Repeatable | Can repeat previously mastered tasks. |
| 3: Defined | Process characterized, fairly well understood |
| 4: Managed | Process measured and controlled. |
| 5: Optimized | Focus on process improvement. |

In short, CMM does not specify what an organization's process should be. Rather it states that organization should have a process, follow it, measure how well it is working, and continually improve it. Due to the competitive nature of software engineering, an organization's CMM level can be a very sensitive piece of information. It is often only disclosed to a client in a confidential bidding situation. It is becoming increasingly common for large RFP's to specify a minimum CMM certification level.

### 2.8.3 Process Improvement

The ability to evaluate a process and adjust it accordingly is a key indicator of an organization's process "maturity". This feedback loop is referred to as process improvement. A practical exercise in process improve taken from the system's development is discussed below.

As discussed in section 2.2 above, the MDA system has undergone three major releases (A, B, and C). In an effort to reduce defect densities experienced in releases A and B, a formal review process was introduced for release C. The details of this process are described in this section. The effectiveness of this process is analyzed and discussed in Chapter 4.

Releases A and B had very little process in support of implementation activities. Formal work package descriptions were created and assigned to the responsible programmer. However, no formal review or closure activities were performed. The programmer's word was generally accepted as sufficient assurance that coding activities had been properly completed. Problems related to system integration and missing functionality were often not discovered until much later, sometimes during formal testing activities.

There is a large body of evidence in the literature to suggest that code inspection is a cheaper and more effective method of discovering defects that formal testing [Mco95]. This evidence supports the decision by senior project staff to introduce a formal review process.

The "Release C" software development process introduced a three-stage review intended to establish the correctness of a work package in terms of standards compliance and general usability. A corresponding checklist was created to

support each stage of the process. The process and workflow are described in the following table and flowchart.

Table 2-6 Review Process Description

| Review Stage | Description |
|---|---|
| Code Review | Team Leader reviews code for standards compliance, readability, and general correctness. Static analysis is not included as part of the review process. |
| Demonstration | Programmer demonstrates new system capability to senior project members. The demonstration is a high level inspection intended to establish general correctness. Workflow, requirements coverage, and usability are emphasized. Formal verification is not performed. |
| Integration & Selloff | Establishes closure by verifying that all code review action items have been completed, requirements have been updated, and common integration traps and pitfalls have been considered. |

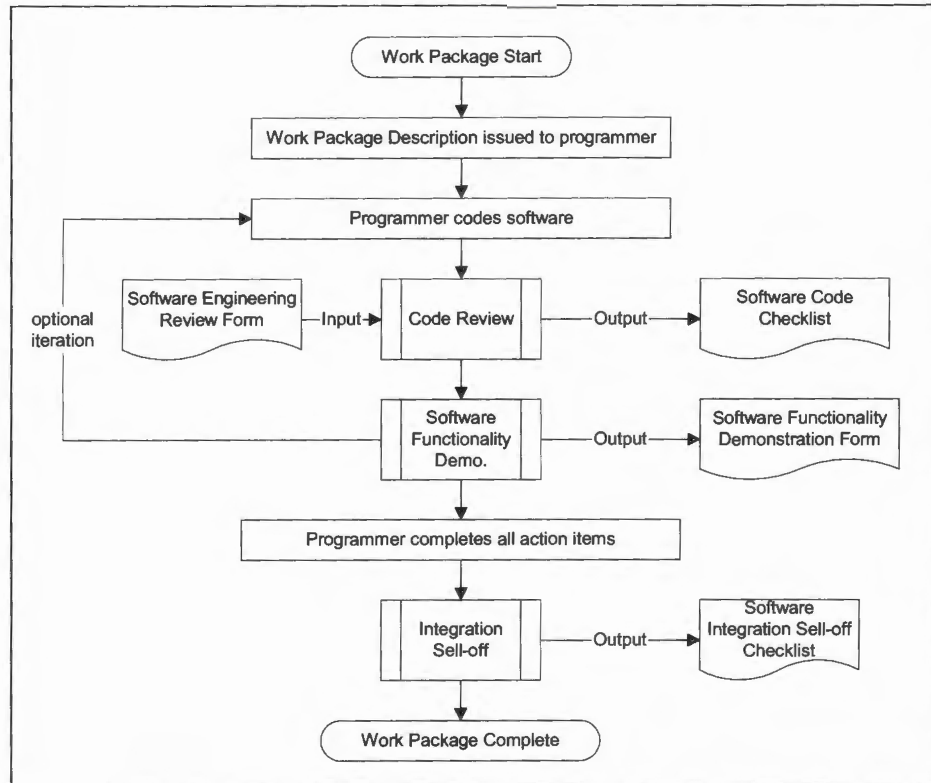


Figure 2-5: Review Process Flowchart

### 2.8.4 Process Engineering

In the McGill-IBM Project on Software Process, Madhavji argues that most software process models are too general to be of any practical value. He uses the term "Software Process Engineering" to refer to the more detail oriented, tangible treatment of the topic [MKL91]. Madhavji's most relevant assertion refers to the relationship between software process and tool support: "*Without adequate process understanding, some aspects of supporting technology cannot be effective. Similarly, without appropriate supporting technology, some aspects of the process are difficult to understand.*" The code review process outlined above is a case in point. It contains a "Code Review" item, which at a high level seems reasonable. However, without appropriate tool support (e.g. effective automated code inspection), the process

cannot be effectively implemented. Conversely, an expensive static analysis tool is ineffective if it is not used as a component of a larger software process. While these statements may appear obvious, the findings documented in Chapter 4 suggest that the integration of software process and tool support is difficult to achieve.

# Chapter 3

# Methodology

Two distinct datasets were collected during this investigation. The first set contains memory leak data identified during dynamic and static analysis. The second contains data on breaking changes collected during the compiler upgrade discussed in section 2.6.4. Each of these datasets and their associated metrics are discussed separately in this section.

## 3.1 Memory Leak Detection

Memory leaks were identified using both dynamic and static analysis techniques. Both approaches have strengths and weaknesses that are discussed in Chapter 4.

Each memory leak detected was recorded and categorized according to type, subsystem, and seniority of the responsible programmer. The programmer responsible for a given memory leak was identified using the project's version control system. The offending file revision was identified using bisection of the revision history.

### 3.1.1 Dynamic Analysis

Dynamic analysis was conducted using the system's automated regression testing capability in conjunction with Microsoft's debug heap facility. As part of the normal development cycle, core portions of system functionality are automatically regression tested. This testing checks for system crashes and differences in

expected results. The regression-testing tool uses a database of test cases intended to achieve broad code coverage. The system's regression testing database contains 2209 test cases. A representative subset containing 1671 of these cases was selected and used as the basis for dynamic analysis.

#### 3.1.1.1 Code Coverage

An attempt was made to quantify the extent of code coverage achieved by the regression-testing database. However, the code base and degree of modularization proved too large for available profiling tools.

#### 3.1.1.2 Instrumentation

After reviewing available memory leak detection tools the native support provided by Microsoft's debug runtime libraries was selected. While not the most user friendly, these proven facilities are available at no additional cost to the existing development environment. Use of Microsoft's debug heap facilities requires that each source file be instrumented with a short pre-processor directive:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

This macro replaces calls to the standard C++ new operator with a special version that records the file name, line number, and size of each allocation. At any point during program execution, a system call can be made to report dynamic allocations that have not yet been freed.

#### 3.1.1.3 Detection Methodology

Debug heap memory leak detection facilities were validated though the deliberate creation of various defects. Following validation of the general approach, the code base was fully instrumented as described above. Analysis of the regression test subset proceeded as follows:

1. Enable debug heap allocations at program start
2. Execute regression test case(s)
3. Dump leaked memory at program shutdown
4. Examine dump file and document leaks

This process was repeated over several weeks until the entire subset of regression tests had been executed. Identification and documentation of leaks in this manner was extremely time consuming. While the causes of some memory leaks are obvious, others are extremely subtle. This challenge was complicated by the lack of call stack information in the debug heap output report.

### 3.1.2 Static Analysis

Static analysis was conducted on the same core subsystems covered during dynamic analysis. The C++test tool produced by Parasoft was employed. This tool implements a wide range of industry standard rule sets. In addition it provides a rule editor that allows the user to define custom rules. Rec. 58 from the classic Ellemtel coding standards document was selected [Ele92]. This rule states simply:

*Do not allocate memory and expect that someone else will de-allocate it later.*

The C++test tool implements Ellemtel Rec. 58 as three separate rules as follows:

If local memory in:

- a global function is allocated via new it should be deleted in this function.
- a class is allocated via new it should be de-allocated in destructor via delete.
- a class is allocated via new a destructor should be defined as well.

The spirit of the Ellemtel rule is sufficiently general to cover all scenarios. Unfortunately the specific implementation of this rule by Parasoft was too simplistic and naïve to be of much use. The Parasoft rules properly track allocations via new and de-allocations via delete in the destructor but falls down when alternative methods of allocation or clean up (e.g. template functions) are used. In these cases the rules are triggered improperly and false violations are reported. Some attempt was made to compensate for this by customizing the rules to account for the specific coding style in question. The rule engine does provide a rich grammar to describe a wide range of syntactic constructs. However, considerable effort would be required to craft a custom rule set that was sufficiently rich to correctly identify the majority of statically detected memory management defects.

To the extent that it was able to correctly identify simple memory leaks, the static analysis tool did make some contribution to the investigation. However, the impact of this approach was not as great as was hoped. This limited experiment involving static analysis has shown that considerable effort is required to create a usable set of rules.

## 3.2 Memory Leak Classification

Numerous defect classification schemes have been proposed. Some, such as Orthogonal Defect Classification (OCD) from IBM are comprehensive [CBC+92]. Other schemes are more focused on a particular class of defect, such as memory leaks [VS04][SC91]. A survey of the literature did not reveal a classification scheme that would describe a memory leak both in terms of a class life cycle and variable context. As such, a two part memory leak classification scheme is proposed here. The first part describes the leak in the context of the class life cycle. Class life cycle is divided into four phases: Initialization, Implementation, Interface, and Destruction. The relationship between a memory leak and the lifecycle of the affected class is highly relevant. Understanding where in the class life-cycle most leaks occur can help to focus preventative action (e.g. static analysis, code review) to achieve the greatest impact. Table 3-1 below provides a definition of each category and examples of the types of leaks that can occur in each one.

Table 3-1 Life-cycle Leak Classification Categories

| Classfication | Description |
|---|---|
| Initialization | Errors related to object creation and assignment.<br>Example 1: missing pointer initialization<br>Example 2: faulty assignment operator |
| Implementation | Errors related to the core implementation of a particular module.<br>Example 1: a pointer is allocated but not deleted.<br>Example 2: a pointer already is use is overwritten |
| Interface | Errors related to poorly defined or understood class interfaces.<br>Example 1: The interface of the function specifies that the pointer variable is deep copied. The caller assumes the pointer is shallow copied and fails to call delete.<br>Example 2: a function interface specifies return by reference. No corresponding member variable exists, so a reference to a heap allocated local variable is returned. |
| Destruction | Covers errors related to object destruction.<br>Example 1: memory is allocated using a member variable pointer with no corresponding delete in either the destructor or a reset method called by the destructor.<br>Example 2: no destructor defined. |

The second part of the classification scheme describes the context of the variables involved. Variable context can be either scalar or composite. Scalar context refers to a single local or member variable. Composite context refers to one or more local or member variables stored within a composite structure such as a list, vector, or struct. The scalar versus composite context of a leak is important as it can help to highlight data types and structures that may be poorly understood by application programmers.

Together the two parts of the classification scheme describe a memory leak. For example, the results of this investigation found that 32% of memory leaks were related to class destruction. Of these 70% involved variables in scalar context; while the remaining 30% involved variables in composite context. The classification scheme described here is used in the experiments described in Chapter 4.

## 3.3 ANSI C++ Compliance

The code base was migrated from VC6 to VC8 (see section 2.6.4 for details) one subsystem at a time. Each breaking change encountered was documented and categorized. In some cases the broken code was benign, while in others the more standard compliant compiler had detected code that was legitimately incorrect. Although this work forms a relatively minor part of the overall investigation these errors are highly instructive. They offer tangible evidence of the relationship between the standards compliance of a compiler and safety of the object code it produces. The nature and distribution of these errors are discussed in Chapter 4.

## 3.4 Metrics

### 3.4.1 Memory Leak Density

Memory leak density is computed as the number of memory leaks detected in a given subsystem divided by the size of that subsystem. Memory leak density is reported as the number of leaks per 1000 source lines of code.

### 3.4.2 Complexity Measurement

McCabe's cyclomatic complexity was chosen as the most appropriate complexity metric for this investigation. The popularity of this metric in the literature and wide tool support were the primary reasons for this choice.

The CCCC (C++ Cyclomatic Complexity) tool was used to gather complexity data. The tool was written by Tim Littlefair as part of his PhD thesis [Lit01]. CCCC computes various code metrics including McCabe's cyclomatic complexity and source line counts.

Raw complexity data was transformed to the "Complexity Density" metric proposed by Gill & Kemerer [GK91]. Complexity density is computed by dividing a subsystem's cyclomatic complexity by its size in source statements. This transformation normalizes complexity values and permits subsystems of differing sizes to be compared. Complexity counts tend to be strongly correlated with module size. This makes sense, as larger modules have more code and hence more logic. Complexity metrics are sometimes discounted as indicators of problematic code in favour of module size [OWB04]. However, when the concept of complexity density is introduced, the value of complexity metrics as a predictor is re-established. The findings presented in Chapter 4 support this assertion.

### 3.4.3 Programmer Experience

The experience level of the programmer responsible for creating each memory leak was classified as either Junior or Senior based on their ranking within the firm. MDA uses a seven level ranking scheme for engineering staff. Programmers at levels 1 to 3 were classified as Junior; programmers at levels 4-7 were classified as Senior. Due to the project duration, some programmers moved from junior to senior positions part way through the implementation. As such, the experience level assigned to a given memory leak is the experience level of the responsible programmer *at the time the leak was created.* Banker et al. have reported a correlation between programmer experience and error rates [BDK+02].

# Chapter 4

# Experiments and Results

This chapter provides details and results of six separate experiments conducted during the course of this investigation. The first experiment involves an analysis of the type and distribution of memory leaks encountered. The second experiment compares memory leak density with subsystem complexity. The third experiment conducts a complementary investigation of the relationship between general defect density (as opposed to memory leak density) and subsystem complexity. The fourth experiment compares memory leaks with programmer experience. The fifth experiment examines the impact of process improvement on general defect density. The sixth experiment presents an analysis of ANSI compliance related breaking changes identified during a C++ compiler upgrade.

## 4.1 Memory Leak Distribution

The goal of this experiment was to quantify memory leak distribution using the classification system proposed in section 3.2. Figure 4-1 below illustrates that majority of memory leaks identified relate to class implementation. This is not surprising as a class implementation typically contains the majority of the code and the associated complexity. More interesting though, is the proportion of leaks related to class destruction and the breakdown of scalar versus composite variable context among them. Almost a third of leaks encountered are associated with class destruction. On one hand this finding is troubling, as destructor logic tends

to be quite straightforward. It is encouraging, however, to realize that by focusing on a single functional area a large positive impact can be realized. Experience gained during this investigation showed that class destruction is one area that lends itself well to static analysis.

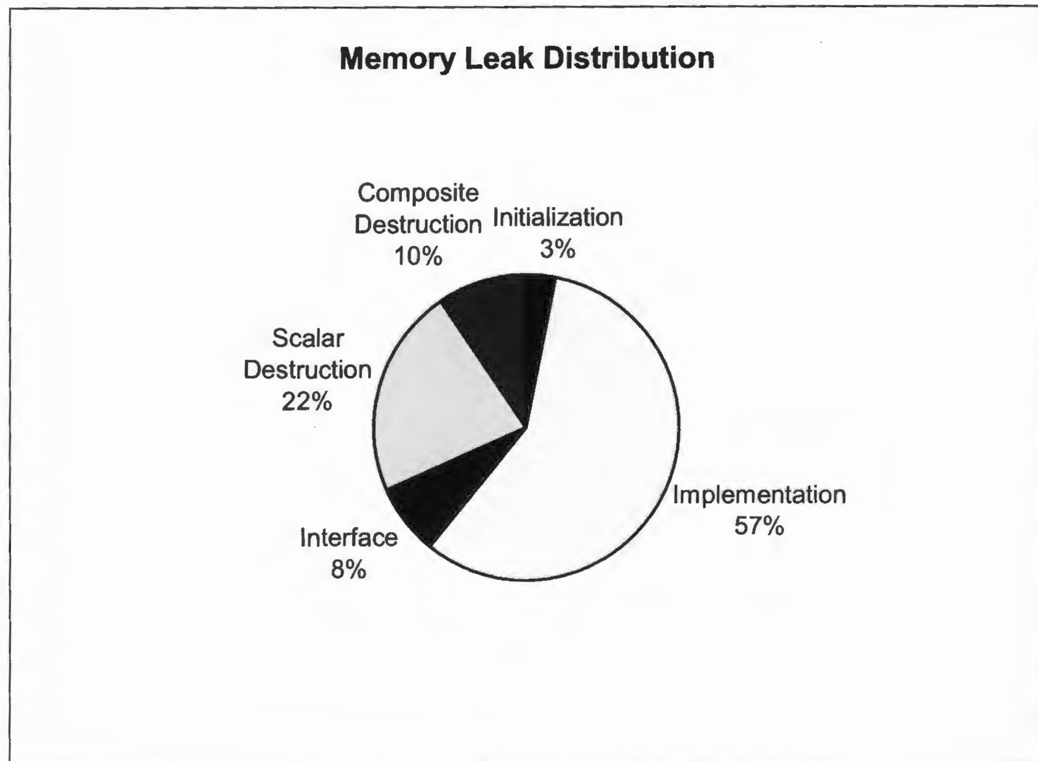


Figure 4-1 Memory Leak Distribution

Interestingly, one third of destruction related leaks involved composite storage. Leaks of this type typically consisted of lists or structs of pointers that were not properly de-allocated. These findings suggest that additional care must be taken when using data structures of this type. As well, it is possible that static analysis techniques could be employed to reduce this particular class of error.

The most significant discovery of this experiment was that 10% of all memory leaks involved the unnecessary use of pointers. The term "unnecessary" must be

qualified: in this experiment, pointer usage was deemed unnecessary if the pointer in question was local to the function and the associated allocation was not needed once the function exited. In other words, a local, stack allocated variable would have been equivalent. This finding suggests that pointer usage should be more tightly constrained, perhaps even to the point of forbidding pointer usage unless otherwise authorized. Indeed, some coding standards prohibit the use of dynamic memory all together [MISRA04].

## 4.2 Memory Leak Density & Subsystem Complexity

The goal of this experiment was to establish if a statistically significant correlation exists between memory leak density and subsystem complexity.

### 4.2.1 Dataset

The dataset consists of 168 memory leaks identified using the static and dynamic analysis techniques. Leaks were identified in 29 of 48 subsystems analyzed. Data on the size and McCabe cyclomatic complexity of each subsystem was collected and used to compute complexity density values. All data was collected using the tools and techniques described in Chapter 3. The dataset shown below contains complexity density and leak density data along with associated mean and difference values to support correlation analysis. Field descriptions for the dataset are described in Table 4-1 below.

Table 4-1 Dataset field descriptions

| Field | Description |
|---|---|
| Size | SLOC count |
| Complexity | Cumulative McCabe complexity |
| Leaks | Number of memory leaks detected |
| Complexity Density | Complexity / Size |
| Leak Density | (Leaks / Size) x 1000 (KSLOCs) |
| Rank X, Rank Y, D, $D^2$ | Spearman non parametric analysis values |

Table 4-2 Complexity Density versus Leak Density Correlation Data

| Subsystem | Size | Complexity | Leaks | Complexity Density (x) | Leak Density (y) (per KSLOC) | Rank x | Rank y | D | D^2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 79562 | 14453 | 14 | 0.1817 | 0.1760 | 28 | 14 | 14 | 196 |
| 2 | 25301 | 2666 | 4 | 0.1054 | 0.1581 | 8 | 10 | -2 | 4 |
| 3 | 5545 | 776 | 1 | 0.1399 | 0.1803 | 17 | 15 | 2 | 4 |
| 4 | 15862 | 1580 | 2 | 0.0996 | 0.1261 | 4 | 8 | -4 | 16 |
| 5 | 23455 | 2466 | 4 | 0.1051 | 0.1705 | 7 | 13 | -6 | 36 |
| 6 | 2583 | 439 | 1 | 0.1700 | 0.3871 | 26 | 25 | 1 | 1 |
| 7 | 82592 | 10892 | 14 | 0.1319 | 0.1695 | 13 | 12 | 1 | 1 |
| 8 | 38839 | 7858 | 1 | 0.2023 | 0.0257 | 29 | 2 | 27 | 729 |
| 9 | 39782 | 2397 | 2 | 0.0603 | 0.0503 | 1 | 3 | -2 | 4 |
| 10 | 9397 | 1265 | 2 | 0.1346 | 0.2128 | 14 | 18 | -4 | 16 |
| 11 | 21627 | 2348 | 3 | 0.1086 | 0.1387 | 9 | 9 | 0 | 0 |
| 12 | 9915 | 1006 | 3 | 0.1015 | 0.3026 | 5 | 22 | -17 | 289 |
| 13 | 36688 | 4170 | 15 | 0.1137 | 0.4089 | 11 | 26 | -15 | 225 |
| 14 | 25014 | 4279 | 3 | 0.1711 | 0.1199 | 27 | 7 | 20 | 400 |
| 15 | 3400 | 459 | 1 | 0.1350 | 0.2941 | 15 | 21 | -6 | 36 |
| 16 | 93149 | 9739 | 1 | 0.1046 | 0.0107 | 6 | 1 | 5 | 25 |
| 17 | 70897 | 9299 | 14 | 0.1312 | 0.1975 | 12 | 17 | -5 | 25 |
| 18 | 1712 | 260 | 1 | 0.1519 | 0.5841 | 21 | 28 | -7 | 49 |
| 19 | 20793 | 3336 | 6 | 0.1604 | 0.2886 | 24 | 20 | 4 | 16 |
| 20 | 16870 | 2329 | 4 | 0.1381 | 0.2371 | 16 | 19 | -3 | 9 |
| 21 | 11864 | 1096 | 1 | 0.0924 | 0.0843 | 2 | 5 | -3 | 9 |
| 22 | 12635 | 1372 | 2 | 0.1086 | 0.1583 | 10 | 11 | -1 | 1 |
| 23 | 23737 | 2230 | 2 | 0.0939 | 0.0843 | 3 | 4 | -1 | 1 |
| 24 | 33359 | 4928 | 14 | 0.1477 | 0.4197 | 18 | 27 | -9 | 81 |
| 25 | 25026 | 3827 | 3 | 0.1529 | 0.1199 | 22 | 6 | 16 | 256 |
| 26 | 50976 | 8109 | 19 | 0.1591 | 0.3727 | 23 | 24 | -1 | 1 |
| 27 | 57263 | 8469 | 11 | 0.1479 | 0.1921 | 19 | 16 | 3 | 9 |
| 28 | 16155 | 2740 | 13 | 0.1696 | 0.8047 | 25 | 29 | -4 | 16 |
| 29 | 3030 | 459 | 1 | 0.1515 | 0.3300 | 20 | 23 | -3 | 9 |
| **Total** | | | | | | | | | **2464** |

### 4.2.2 Experimental Details and Results

Analysis was performed on the dataset to determine whether a statistically significant correlation exists between complexity density and leak density. Spearman non-parametric analysis was selected in order to avoid making assumptions regarding the distribution of data [JB92].

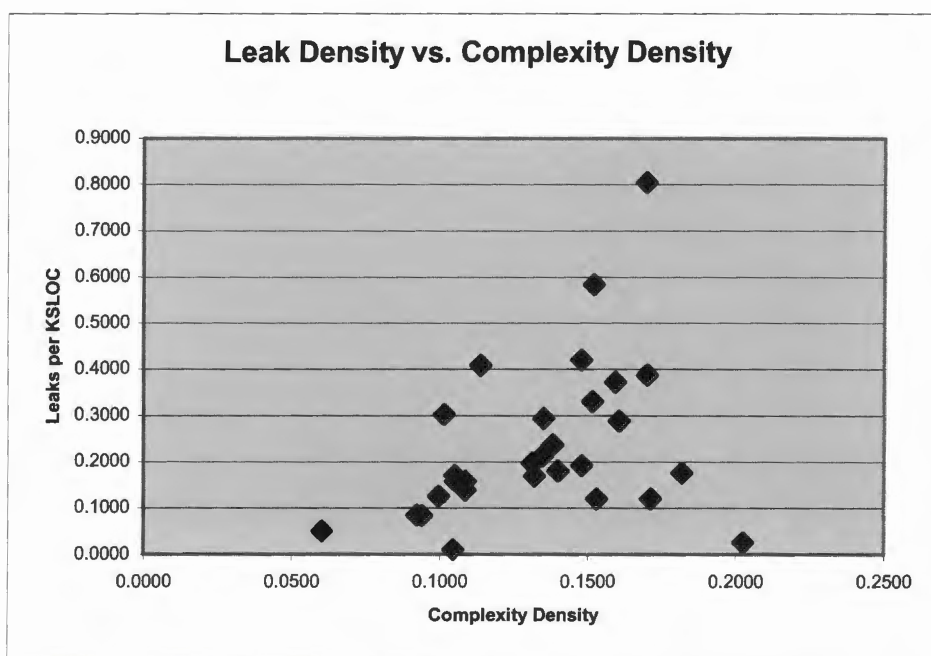


Figure 4-2: Leak Density vs Complexity Density Scatter Plot

A scatter plot of the data show above revealed a potentially linear relationship between the variables in question. As the complexity density (x-axis) increases, there is an apparent increase in the leaks per KSLOC (y-axis).

#### 4.2.2.1 Spearman Analysis

Spearman non-parametric correlation calculations for the complexity density versus leak density experiment are shown below:

$H_0$ = There is no correlation between complexity density and leak density.

$$r_s = 1 - \frac{6\sum D^2}{N(N^2-1)} = 1 - \frac{6 \times 2464}{29(29^2-1)} = 0.39$$

The computed $r$ value of 0.39 exceeds the critical r value of 0.37 ($p = 0.05$) for df = 27 (two tailed).

These results allow us to reject $H_0$ at $\alpha = 0.05$ and suggest a medium strength correlation between complexity density and leak density.

#### 4.2.2.2 Discussion

The Spearman test shows a medium strength correlation ($r = 0.39$; $\alpha = 0.05$). Clearly there are many determinants of leak density in addition to program decision complexity. Having said this, this result shows that a statistically significant relationship exists between the two parameters. These findings are consistent with similar research conducted by Nagappan et al [NBZ06].

## 4.3 Defect Density & Subsystem Complexity

As a complement to the Leak Density & Subsystem Complexity experiment, a secondary investigation was conducted to see if a relationship existed between general (e.g. not memory leak specific) defect density and subsystem complexity. Data used in this experiment was gathered from the system's defect tracking database. A subsystem was implicated in a defect if at least one of its files was changed during the course of the defect's resolution. The defect count per subsystem was converted to a density value. Complexity density values were computed as described above. An initial scatter plot of the data shown in Figure 4-3 below showed no discernable relationship between the two parameters.

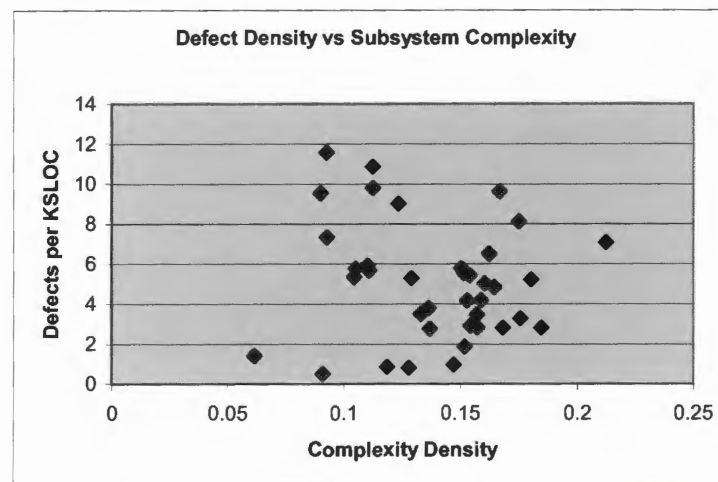


Figure 4-3: Defect Density vs Subsystem Complexity

Subsequent numerical analysis using the Spearman technique confirmed this lack of correlation ($r \approx -0.1$). This finding is consistent with those of both Gitten and Ostrand, both of whom found no signification relationship between subsystem complexity and defect density [GKG05][OWB04].

## 4.4 Memory Leaks & Programmer Experience

### 4.4.1 Dataset

This experiment utilized the same data set as the experiment described above. For each memory leak in the dataset a responsible programmer was identified. and classified as either junior or senior. Of the 168 memory leaks identified, 101 were created by junior programmers, while the remaining 67 were created by Senior programmers. 24 junior programmers and 10 senior programmers are represented in the dataset. All data was collected using the tools and techniques described in Chapter 3.

### 4.4.2 Experimental Details and Results

Analysis was performed on the dataset to determine whether a statistically significant relationship exists between the number of memory leaks created and the level of programmer experience. Due to the categorical nature of the data a Chi-Squared ($X^2$) analysis was selected [JB92].

#### 4.4.2.1 Chi-Squared Analysis

Chi-Squared calculations for the programmer experience versus memory leak experiment are shown below:

$H_0$ = There is no relationship between programmer experience and the creation of memory leaks.

Table 4-3 Chi Squared Analysis of Programmer Experience

| | Junior | Senior | Total |
|---|---|---|---|
| Actual Leaks (O) | 101.00 | 67.00 | 168.00 |
| Expected Leaks (E) | 118.59 | 49.41 | 168.00 |
| O – E | -17.59 | 17.59 | 168.00 |
| $(O - E)^2$ | 309.35 | 309.35 | |
| $(O - E)^2 / E$ | 2.61 | 6.26 | 8.87 |

Expected Leak values were computed by multiplying the total leaks found by the proportion of Junior and Senior programmers on the project. For example, approximately 70% of programmers are Junior level, as such the expected number of leaks for Junior programmers is 70% of 168, or 118.59.

The computed $X^2$ value of 8.87 exceeds the critical value of 6.64 ($p = 0.01$) for 1 degree of freedom. This result allow us to reject $H_0$ at $\alpha = 0.01$ and indicates that senior programmers created proportionally more memory leaks than expected.

#### 4.4.2.2 Discussion

This somewhat counter intuitive result may have a variety of causes. It is possible that senior programmers are more comfortable with the advanced topic of dynamic memory management. Given this they may be more likely to use the technique and hence fall victim to its associated pitfalls. A second possible explanation is that senior programmers are more productive than their junior

counterparts. As such, senior programmers may produce a greater absolute number of errors even their error rate is lower. Training issues must also be considered. Although senior programmers may have more software development experience in the broad sense, they may be less familiar with newer object oriented techniques.

## 4.5 Process Improvement & Defect Density

This experiment utilized the project's defect management database to assess the impact of the process improvements discussed in section 2.8. Defect densities of software written before and after the enactment of the review process were compared. The defect densities discussed here are not restricted to memory management problems. All pre-release defects from Release B and C are compared. Post release defects are not considered, as Release C post release defect data was not available at the time of writing.

Table 4-4 Release B vs. C Defect Densities

| Category | Defects / KSLOC |
| --- | --- |
| Release B Pre-fielding | 4.49 |
| Release C Pre-fielding | 5.07 |

It is clear from these figures that the process improvement measures introduced between Release B and C did not help to reduce defect densities. Having said this some qualification is required. Part of the strategy behind the review process was to identify defects earlier during the development cycle. In effect, turning post-release defects into pre-release defects so they can be dealt with more easily and cheaply. As such, it is possible that this effect is occurring here, as Release C pre-release defects are indeed higher than in Release B. This picture will not be complete until Release C post-release defects are available. Even so, a defect is a defect, whether it occurs pre-release or post-release. While it is good to have

identified it, it would be better had it not occurred at all. The goal of the software review process described here was to prevent defects from occurring through improved understanding of requirements, code inspection, and peer review demonstrations. Given these stated objectives, it is hard to argue that the process was successful in the face of the numbers presented here.

The results presented throughout this investigation point to the need for a comprehensive white-box testing program. Such a program would incorporate both static and dynamic analysis techniques into the weekly build cycle of the project. The project's current black-box testing program is comprehensive, but it is only part of the picture.

The existing regression automated regression and black-box testing regimes must be supported by a parallel white-box program. Each type of testing emphasizes a specific category of errors: Black-box testing is requirements focused, while regression testing detects system crashes and unexpected behaviour changes. White box testing can detect an entirely different category of defects, and to a certain extent, can do so in a pre-emptive manner. Memory management, logical errors (e.g. logical incompleteness, orphaned branches, etc), and complexity hot spots can all be identified using static and dynamic analysis techniques. Without a white-box regime, the testing program is simply not as complete as it could be.

## 4.6 ANSI C++ Compliance

This experiment examines the distribution of errors resulting from the compiler upgrade described in section 2.6.4. This upgrade yielded approximately 3300 compiler warnings and errors. Once spurious and informational warnings were reconciled, 1598 meaningful warnings and errors remained. From this set, 66 legitimate software defects were identified. The following table shows the proportion of a particular type of error or warning that yielded a legitimate defect.

Table 4-5 Software defects identified by compiler error

| Error Number | Error Count | Defect Count | Proportion | Description |
|---|---|---|---|---|
| C2065 | 170 | 3 | 0.02 | Undeclared identifier |
| C2440 | 110 | 2 | 0.02 | Type conversion |
| C2675 | 1 | 1 | 1 | Operator resolution |
| C2678 | 2 | 2 | 1 | Operator resolution |
| C3867 | 35 | 35 | 1 | Function call missing arg list |
| C4353 | 1 | 1 | 1 | Non standard expression |
| C4430 | 300 | 22 | 0.07 | Missing type specifier (default int) |

It is interesting to note that some error types appear to have a higher predictive value than others. There also appears to be an inverse relationship between the frequency of an error and its predictive value. For example, error C3867 (function call missing arguments) was 100% predictive, but was only $1/10^{th}$ as frequent as error C4430 (missing type specifier) which was only 7% predictive.

Examples of the unintended behaviour associated with each of the defect types identified above are outlined in the table below.

Table 4-6 ANSI Compliance Error Examples

| Error Number | Affected Behaviour | Example | Description |
| --- | --- | --- | --- |
| C2065 | Loop Execution | `for (int i  = 1; i < 10; i++);`<br>`{`<br>`    foo(i);`<br>`}` | The error here is the semi colon at the end of the loop control statement. This is a classic example of un-intended behaviour [Hat95]. Under the VC6 compiler, the statements inside the loop block would have executed once using the last value of i. |
| C2440<br>C3867 | Assignment | `void foo(void)`<br>`{`<br>`    double x = getDistance;//C3867`<br>`}` | The "getDistance" function is called without providing a parameter list. As such the variable x is assigned the address of the getDistance function rather than its return value. C2678 captures another variant of this defect involving user define rather than built in types. |
| C2675<br>C2678 | Comparison | `enum Status {s_PASS, s_FAIL};`<br><br>`Status x = foo();`<br><br>`if (x == "s_FAIL") //C2678`<br>`{`<br>`    return false;`<br>`}` | Consider the enumerated type "Status" that defines status codes. In the "if" statement above, the enumerated value s_FAILED is incorrectly wrapped in quotes and treated a string. Under the VC6 compiler the code compiled without error, and the block of code in the 'true' branch of the 'if' statement would never execute. |
| C4345 | Undefined | `if (p != NULL (p->isValid()) //C4345`<br>`{`<br>`    return true;`<br>`}` | The error here is a missing logical 'AND' operator && in the 'if' statement. Under the VC6 compiler the code compiled without error. In this case it appears that NULL is being used as a function expression. This would almost certainly result in a crash, or perhaps worse, some undefined behaviour. |
| C4430 | Type conversion | `const x = 5.3; //x actually = 5`<br>`foo(x);` | This is perhaps the most vexing and potentially dangerous coding error found during this investigation. Under the VC6, variables that were declared without a type specifier would be defaulted to int. In the example shown here, the numeric literal 5.3 is implicitly converted to an integer and rounded to 5. |

# Chapter 5

# Conclusions

## 5.1 Contributions

The research presented here is distinct in two important ways. The first is the focus on memory management problems in the context of code complexity. When correlating defects with code complexity, existing research tends not to distinguish between different categories of faults. The second is the use of the complexity density metric, introduced by Gill and Kemerer, in the context of memory management defects [GK91]. In addition, a two-part memory leak classification scheme is described for categorizing a memory leak in terms of both of class life cycle and variable context.

## 5.2 Findings

A statistically significant relationship was found between memory leak density and subsystem complexity. The Spearman (non parametric) rank correlation technique yielded a medium strength correlation ($r = 0.39$, $\alpha = 0.05$).

A complementary investigation focusing on general defect densities did not find a statistically significant relationship between general defects and subsystem complexity. These findings are interesting as they suggest memory management related problems could be sensitive to decision complexity.

An analysis of where memory leaks occurred during the class lifecycle indicated that 57% of leaks occurred in class implementation logic, while 32% occurred in class destructors. Of the destruction related leaks, 30% involved composite storage. Leaks of this type typically consisted of lists or structs of pointers that were not properly de-allocated. These findings suggest that additional care must be taken when using data structures of this type.

On the whole, 10% of all memory leaks involved the unnecessary use of pointers. This finding suggests that pointer usage should be much more tightly constrained, perhaps even to the point of forbidding pointer usage unless otherwise authorized.

A Chi-Squared analysis comparing memory leaks with programmer experience showed that Senior programmers create proportionally more leaks than their Junior counterparts ($\alpha = 0.01$).

The impact of a software process improvement effort was examined. A structured process involving code review, demonstrations, and formal sell-off was found to have no impact on general defect densities. These findings, together with those listed above, are highly suggestive of a need for process improvement. Specifically, a comprehensive white-box testing program is required. Automated software inspection technologies are now sufficiently mature to support this objective.

Coding defects identified as a result of improved ANSI C++ compliance were enumerated and discussed. The frequencies and predictive power of various error types were presented and an inverse relationship was noted between the two parameters.

## 5.3 Future Work

This investigation has laid the groundwork work for a future prospective study. If a comprehensive white-box testing program is introduced, its impact can be measured using many of the same experiments and metrics employed here. Differences from before and after can be measured and the impact of the testing program can be quantified. Additional work in the area of static analysis is required to support this goal.

Additional investigation into the relationship between memory leak density and code complexity is also required. The findings presented here suggest a moderate correlation between these two parameters. Future investigations will help to improve our understanding of this relationship.

# Bibliography


[Ale77] C. Alexander. A Pattern Language. Oxford University Press, 1977.

[Ale01] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Boston, 2001.

[ANSI03] American National Standards Institute. Programming languages - C++, Final edition, 2003.

[ASU98] A. Aho, R. Sethi, J. Ullman. Compilers. Principles, Techniques, and Tools. Addison-Wesley, Reading, 1988

[BDK+93] R. Banker, S. Datar, C. Kemerer, Dani Zweig. Software Complexity And Maintenance Costs. Communications of the ACM, Vol. 36, No. 11, November 1993, pp. 81-94.

[BDK+02] R. Banker, S. Datar, C. Kemerer, Dani Zweig. Software Errors and Software Maintenance Management. Information Technology and Management, Vol. 3, No. 1/2, January 2002, pp. 25-41.

[Bez90] B. Bezier. Software Testing Techniques, 2nd Edition. Van Nostrand Reinhold, New York, 1990.

[Boe78] B. Boehm. Characteristics of Software Quality. North-Holland, Amsterdam, 1978.

[Con86] S. Conte. Software Engineering Metrics And Models. Benjamin/Cummings, Menlo Park, 1986.

[CBC+92] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M Wong. Orthogonal Defect Classification - A Concept for In-Process Measurements, IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992, pp. 943-956.

[Ele92] Ellemtel Telecommunications Systems Laboratories. Programming in C++, Rules and Recommendations. Retrieved on August 18th, 2006 from http://www.chris-lott.org/resources/cstyle/Ellemtel-rules-mm.html

[ESA96] European Space Agency. Ariane 5 Flight 501 Failure Report by the Inquiry Board. Retrieved on August 9th, 2006 from http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

[Eva04] I. Evans. Achieving Software Quality Through Teamwork. Artech House, London. 2004.

[FAA88] Federal Aviation Administration. Advisory Circular 25.1309-1A. U.S. Department of Transportation, 1988.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, 1995.

[GKG05] M. Gittens, Y. Kim, D. Godwin. The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software. Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 2005.

[GK91] G. Gill, C. Kemerer. Cyclomatic Complexity Density and Software Maintenance Productivity. IEEE Transactions on Software Engineering, Vol. 17, No. 12, December 1991, pp. 1284-1288.

[Hal77] M. Halstead. Elements of Software Science. Elsevier, New York, 1977.

[Hat95] L. Hatton. Safer C. Developing Software for High-integrity and Safety-critical Systems. McGraw-Hill, London. 1995.

[Her97] D. Herrmann. Software Safety and Reliability. IEEE Computer Society Press, 1999.

[Joh78] S. Johnson. Lint, A C program checker. Computer Science Tech. report 65. Bell Laboratories, 1978

[JB92] R. Johnson, G. Bhattachary. Statistics: Principles and Methods. Wiley, New York. 1992.

[Kit95] E. Kit. Software Testing In The Real World. Addison-Wesley, Harlow, 1995.

[KS97] C. Kemerer. S. Slaughter. Determinants of Software Maintenance Profiles: An Empirical Investigation. Software Maintenance: Research And Practice, Vol. 9, pp. 235-251, 1997.

[Lar02] C. Larman. Applying UML And Patterns. Prentice Hall, 2002.

[Lev93] N. Leveson. An Investigation of the Therac-25 Accidents. IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41.

[Lip92] S. Lipmann. C++ Primer, 2nd Edition. Addison-Wesley, Reading, 1992.

[Lit01] T. Littlefair. An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment., PhD thesis, Edith Cowan University, June 2001.

[LS93] B. Littlewood, L. Strigini. Validation of ultrahigh dependability for software-based systems. Communications of the ACM, Vol. 36, Issue 11, November 1993, pp. 69-80.

[Mar92] D. Marks. Testing Very Big Systems. McGraw-Hill, 1992

[Mca76] T. McCabe. A Complexity Measure. Proceedings of the 2nd international conference on Software engineering (ICSE). San Francisco, California, United States. IEEE Computer Society Press, 1976.

[Mco93] S. McConnell. Code Complete. A Practical Handbook of Software Construction. Microsoft Press, Redmond, 1993.

[Mey98] S. Meyers. Effective C++. 2nd Edition. Addison-Wesley, Boston. 1998.

[MISRA04] Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems. MIRA Ltd. Retrieved from http://www.misra.org.uk on August 23rd, 2006

[MKL91] N. Madhavji, K. Toubache, E. Lynch. The IBM-McGill Project on Software Process. Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research. October 1991.

[MR94] J. Miller, G. Rozas. Garbage Collection is Fast, But a Stack is Faster. MIT Artificial Intelligence Memo 1462, March 1994.

[MS06] Microsoft Developer Network Library. Common Problems When Creating a Release Build. Retrieved August 9, 2006 from http://msdn2.microsoft.com/en-us/library/dykf6bx9.aspx

[NBZ06] N. Nagappan, T. Ball, A. Zeller. Mining Metrics to Predict Component Failures. The 28th International Conference on Software Engineering (ICSE'06), 2006.

[NWH+04] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, M. Vouk. Preliminary Results On Using Static Analysis Tools For Software Inspection. The 15th International Symposium on Software Reliability Engineering (ISSRE'04). 2004.

[OWB04] T. Ostrand, E. Weyuker, R. Bell. Where the bugs are. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04). 2004.

[Pet99] C. Petzold. Programming Windows. Microsoft Press, Redmond, 1999.

[Ris98] L. Rising. The patterns handbook : techniques, strategies, and applications, collected and introduced by Linda Rising. Cambridge University Press, 1998.

[Sta94] The Standish Group. The CHAOS Report (1994). Retrieved August 9th, 2006 from http://www.standishgroup.com/sample_research/chaos_1994_1.php

[SEI02] Software Engineering Institute. Carnegie Mellon University. Capability Maturity Model Integration (CMMI), Version 1.1. Retrieved on August 22nd, 2006 from http://www.sei.cmu.edu/cmmi/models

[SEI06] Software Engineering Institute. Carnegie Mellon University. Cyclomatic Complexity. Retrieved on August 15th, 2006 from http://www.sei.cmu.edu/str/descriptions

[Som91] I. Sommerville. Software Engineering, 6th Edition. Addison-Wesley, Harlow, 2001.

[Str97] B. Stroustrup. The C++ Programming Language. Addison-Wesley, Boston, 1997.

[SC91] M. Sullivan, R. Chillarege. Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems. Proceedings of the 21st International Symposium on Fault Tolerant Computing, 1991.

[VS04] H. Verta, T. Saridakis. Detection of Heap Management Flaws in Component-Based Software. Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), 2004.

# Appendix 1: Non Disclosure Agreement

This section contains a signed copy of a non disclosure agreement between UNBC and MDA. This agreement governs the usage of the MDA intellectual property upon which this investigation is based.

# MUTUAL CONFIDENTIAL DISCLOSURE AGREEMENT

THIS AGREEMENT made this 01 day of June in the year 2006 (the "Effective Date").

BY AND BETWEEN:

**MacDONALD, DETTWILER AND ASSOCIATES LTD.**
a company duly incorporated under the laws of Canada,
having offices at 13800 Commerce Parkway, Richmond, British Columbia, Canada V6V 2J3,

("MDA")

AND:

**UNIVERSITY OF NORTHERN BRITISH COLUMBIA**
3333 University Way,
Prince George, B.C.
V2N 4Z9

In consideration of the mutual premises and agreements herein contained, and other good and valuable consideration, the receipt and sufficiency of which is hereby acknowledged, the parties agree as follows:

1. "Confidential Information" shall mean any information which is confidential and proprietary information of each party concerning each party's technical, scientific and business interests not generally available to third parties consisting of but not limited to: (i) software (source and executable or object code), algorithms, computer processing systems, techniques, methodologies, formulae, processes, compilations of information, drawings, proposals, job notes, reports, records, and specifications, and related documentation in any media, including all modifications, enhancements, updates and derivatives; (ii) unique software and hardware configurations, design concepts and all materials developed therefrom; (iii) business plans, customer contacts, licenses, the prices each party obtains or has obtained for its software, products or services and any other materials or information relating to the business of each party or each party's good will, each party's subsidiaries, owners, affiliates and divisions or any of each party's customers; (iv) any confidential information in any media which is owned by a third party and provided to either party under a confidentiality agreement; (v) trade secrets which derive economic value, actual or potential, from not being generally known to other persons who might obtain economic value from its disclosure or use and is the subject of efforts that are reasonable under the circumstances to maintain its secrecy; and (vi) any other confidential information of each party which is determined by a court of competent jurisdiction not to rise to the level of a trade secret under applicable law.

2. Each party shall reduce to tangible form, mark as proprietary, and provide to the other party, only such Confidential Information relating to the subject matter described at the end of this Agreement (the "Subject Matter") as the parties determine is reasonably required to achieve the Purpose as defined below.

3. All rights, title and interest in and to the Confidential Information shall remain the exclusive worldwide property of the party which provided the Confidential Information except the Confidential Information owned by a third party as set out in Article 1(iv) of this Agreement.

4. Neither party shall, directly or indirectly, use the Confidential Information for the design or creation of any product or service, or use the Confidential Information in any other manner, except as reasonably required for the purpose described at the end of this Agreement (the "Purpose").

5. If the disclosure of the Confidential Information should give rise to a business opportunity to commercially exploit the Confidential Information, any such exploitation by either party, or by either party assisting any third party, or the creation of any product or service which is directly or indirectly based on, derived from, or uses the Confidential Information without the other party's consent, is not permitted.

6. Each party shall keep the Confidential Information of the other party in strict confidence. Neither party shall directly or indirectly disclose, allow access to, transmit, or transfer the Confidential Information of the other party to a third party except to those of its employees, directors, agents, who have an actual need to know the Confidential Information for the Purpose. The recipient shall, prior to disclosing the Confidential Information to such employees and agents, obtain their agreement to receive and use the Confidential Information on a confidential basis on the same conditions as contained in this Agreement. MDA who is contemplating the disclosure of Confidential Information to the University of Northern BC acknowledges that the University by its very nature is an open public research institution with students passing through in an open and uncontrolled manner and therefore cannot provide the same degree of security for its own Confidential Information as that which is customary in an industrial research centre. However, the University will use the same care and discretion to avoid disclosure of Confidential Information as it uses for its own similar Confidential Information that it does not wish to disclose.

7. The Confidential Information shall not be reproduced in any form or stored in a data base, by the recipient without the prior written consent of the other party. All copies of the Confidential Information shall contain only the same proprietary notices, which appear on the original Information.

8. This Agreement shall not apply to Confidential Information which can clearly be proven by documentation to have become readily available to the general public in the same form through no breach of this Agreement, or which was lawfully obtained by the recipient from an independent third party having no confidentiality obligation to either party, or which was in the recipient's possession in fully recorded form prior to date of disclosure by the other party. The burden of providing these exceptions resides with the recipient. This Agreement also applies to (i) Confidential Information provided to either party even if such Confidential Information could be, or was subsequently, obtained by reverse engineering, and (ii) Confidential Information received by either party prior to this Agreement being signed.

9. This Agreement shall not constitute any representation, warranty or guarantee to either party with respect to the value of the information to the recipient or that the Confidential Information does not infringe any rights of third parties. Neither party shall be held liable for any errors or omissions in the Confidential Information, or use of the Confidential Information.

10. The entering into of this Agreement shall not constitute any obligation on the part of either party to enter into any further agreement with the other party.

11. Both parties recognize that each party may be engaged in the development of hardware or software products or other technology or services which may be competitive with those of the other party to this Agreement, and nothing in this Agreement shall be construed to prohibit either party from engaging in the research, development, marketing, sale or licensing of any product which is independently developed and produced without the use of the other party's Confidential Information.

12. Each party shall upon completion of the Purpose or upon the request of the other party, whichever first occurs, immediately return to the other party the Confidential Information and all copies thereof in all forms, and permanently delete the Confidential Information from all retrieval systems and data bases in which it may be found.

13. The term of this Agreement shall commence on the Effective Date and shall continue for the period of time set out at the end of this Agreement and notwithstanding the foregoing, neither party shall disclose or utilize trade secrets as set out in Article 1(v) of this Agreement for an unlimited period of time.

14. In the event of a breach by either party of any provision of this Agreement, the other party shall, in addition to and not in substitution for any other remedy available to it in respect of such breach, be entitled to injunctive relief which restrains the party in breach from committing or continuing such breach.

15. If it is held by a court or other lawful authority of competent jurisdiction that any provision of this Agreement or part thereof is void, illegal, invalid or unenforceable then such provision or part shall be deemed stricken, and the remaining provisions shall be severable and remain valid, in full force and effect.

16. Neither party shall assign this Agreement or any rights or obligations under this Agreement without the prior written consent of the other party, and any attempt to do so without consent shall be null, void and of no effect.

17. This Agreement shall be governed by, subject to, interpreted and enforced in all respects in accordance with the laws which apply in the province of British Columbia, Canada and the parties shall submit to the exclusive jurisdiction of the courts of British Columbia.

18. Each party shall comply and shall cause its affiliates to comply with all applicable laws and regulations pertaining to export or re-export of Confidential Information to the other party or its affiliates.

19. No waiver of any provision of this Agreement or of any breach of this Agreement shall be effective, unless such waiver is in writing and signed by the party providing such waiver. Any signed waiver shall not operate or be construed as a waiver of any other provision or any other breech of this Agreement.

20. Neither party shall make use of any discussions concerning this Agreement, or this Agreement itself, for publicity, advertising or marketing, or disclose that either party has entered into this Agreement, without the prior written consent of the other party.

21. This Agreement constitutes the entire agreement and understanding between parties and supersedes all prior discussions and agreements between the parties hereto relating generally to the same subject matter.

**Subject Matter:** Master's Thesis from Stephen Wickham regarding a retrospective analysis of the GPD code base will be conducted. Coding errors that cause memory leaks and interfere with the creation of a release build will be measured and corrected. The primary objective of the exercise is the creation of a release build of GPD. Supporting objectives include identification and correction of memory leaks, and preparation of the GPD code base for use with Microsoft Visual Studio 2005. A report discussing the problems encountered and the distribution of errors will be provided.

**Purpose:** To protect business sensitive confidential and proprietary information discussed, developed and/or supplied.

**Term:** Five (5) Years from the Effective Date or three (3) years from the latest date of disclosure of Confidential Information from one party to the other, whichever is later.

MUTUAL CONFIDENTIAL DISCLOSURE AGREEMENT

IN WITNESS WHEREOF, the parties have entered into this Agreement by their duly authorized representatives.

MACDONALD, DETTWILER AND ASSOCIATES LTD.

By: ______________________
(Signature)

Name: Stephen J. Patko
(Printed/Typed)
STEPHEN J. PATKO
DIRECTOR, PROCUREMENT SERVICES

Title: ______________________

Date: 2006 JUN 26

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

By: ______________________
(Signature)

Name: D. Max Blouw
(Printed/Typed)

Title: VP Research

Date: ______________________

Dr. Siamak Rezaei

By: ______________________
(Signature)

Name: Siamak Rezaei
(Printed/Typed)

Title: Associate Professor

Date: 09 July 2006