Design Guidelines and Documentation
Paradigms for Object Oriented Programs
DESIGN GUIDELINES AND DOCUMENTATION PARADIGMS FOR OBJECT ORIENTED PROGRAMS

By

JENS ANDERSEN, B.B.A.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirement

for the Degree

Master of Science

McMaster University

Copyright by Jens Andersen, January 1998
MASTER OF SCIENCE (1998)  
(Computing Science and Systems)  
McMaster University  
Hamilton, Ontario, Canada

TITLE:  
Design Guidelines and Documentation Paradigms For Object Oriented Programs

AUTHOR:  
Jens Andexer,  
B.B.A. (Simon Fraser University, British Columbia, Canada)

SUPERVISOR:  
David L. Parnas

NUMBER OF PAGES:  
vii, 420
Abstract

What’s the Problem with software? Almost daily one reads about the advances in technology and leading the way, it seems, is the computer industry. Storage costs are dropping, processor cycles per dollar are increasing, but nagging at the heels of these successes are stories from the software industry reporting cost overruns and ballooning expenses. Why is that? And what about today’s crop of new software technologies? Surely they can help solve the problems.

Object Oriented programming languages are a new technology. Some believe this new technology can cure many of the software industries ills, that it is a “silver bullet[52]”.

But in order to find a cure one must understand the disease, not just treat the symptoms. Costs are not the disease, only a symptom. But if costs are only a symptom what are their main contributors? What is driving the them up? And can OO deliver what is hoped for?

The answers to these questions do not lay in new cutting edge of technology, their answers lay in what happens to cutting edge technology over time.

This thesis explores what happens to software as it ages. Starting from a basic understanding of software aging, a deeper understanding of its details is established by analysing the most prevalent software maintenance process in industry. This analysis uncovers what makes software maintenance so difficult and, therefore, so expensive.

The information gained from the analysis of the software maintenance process is used to develop design guidelines and a documentation paradigms that take much of the speculation out of designing software for maintenance. Finally, a case study is developed that shows what the documentation for a system developed using this thesis might look like.
Executive Summary

Typically software is developed for its functionality, with thoughts about its maintainability receiving only passing consideration. Developing software for maintainability, however, requires much more than correctly implementing specifications. When developing software for maintainability details about the software’s design and its construction must be considered as well. How the system is broken into parts, how the parts are related to each other, how the programs themselves are crafted, how errors are handled, how errors are reported and how effectively the documentation describes the software are only some of the considerations when developing software for maintainability. By paying attention to these details software can be constructed so that changes can be made with a minimum of effort and so that modified software can be moved to production quickly, while minimising the risk of introducing new errors that lead to a loss of trust in the production system.
Contents

1 Introduction ......................................................... 1
   1.1 Purpose ...................................................... 3
   1.2 Scope ......................................................... 4
   1.3 Related Work ................................................ 4
   1.4 Outline ....................................................... 4

2 Software and The Effects of Time ................................. 7
   2.1 Maintenance-in-the-Large .................................. 7
      2.1.1 Several Maintainers ................................. 9
      2.1.2 By-products of Change ............................. 9
      2.1.3 Risk ..................................................... 10
      2.1.4 Deadlines ............................................. 10
   2.2 Software under Maintenance ............................... 11
      2.2.1 The consequences of modifying software .......... 12
      2.2.2 Software Deteriorates During Maintenance ........ 14
      2.2.3 How Does Software Deterioration Affect Programmers? 16
      2.2.4 How Does Software Deterioration Affect its Owner? 16
   2.3 Is Object Orientation a Remedy for Software Maintenance Woes? 17
      2.3.1 The real-world ....................................... 18
      2.3.2 The programming-world ........................... 21
      2.3.3 Is OO Better for Maintenance? .................. 31
   2.4 Conclusion .................................................. 32

3 Software Maintenance Categories ................................ 35
   3.1 Corrective Maintenance ................................... 35
      3.1.1 Errors that Originate in Software Requirements Specification 36
      3.1.2 Errors that Originate in Implementation ........... 37
   3.2 Adaptive Maintenance ..................................... 38
3.3 Perfective Maintenance ............................................. 38
3.4 Allocation of Maintenance Effort .................................. 38

4 Software Maintenance Process ........................................ 41
4.1 Defining and Understanding the Change ............................ 43
  4.1.1 Difficulties .................................................... 43
  4.1.2 Observations ................................................... 45
4.2 Document Review ......................................................... 45
  4.2.1 Difficulties .................................................... 46
  4.2.2 Observations ................................................... 49
4.3 Program Comprehension ................................................. 49
  4.3.1 Sources of Information ....................................... 51
  4.3.2 Difficulties .................................................... 65
  4.3.3 Observations ................................................... 76
4.4 Implementation ......................................................... 76
  4.4.1 Difficulties .................................................... 77
  4.4.2 Observations ................................................... 87
4.5 Testing ................................................................. 87
  4.5.1 Difficulties .................................................... 90
  4.5.2 Observations ................................................... 94
4.6 Update Documentation ............................................... 95
  4.6.1 Difficulties .................................................... 95
  4.6.2 Observations ................................................... 96
4.7 Conclusion ............................................................. 96

5 Design Guidelines ...................................................... 99
5.1 What to Make a Secret ............................................... 101
  5.1.1 Information Hiding as a Decomposition Criteria ............. 102
  5.1.2 Implications for Maintenance ................................ 107
5.2 How to Tell a Story but Keep a Secret ............................ 107
  5.2.1 Abstract in Small Steps ...................................... 111
  5.2.2 Use Package Access for Inheritance .......................... 111
  5.2.3 One Abstraction per Package ................................ 111
  5.2.4 Use Public Access to Define an Interface ..................... 112
  5.2.5 Use Private Access for Containment ......................... 112
  5.2.6 Nest Virtual Machines ....................................... 113
  5.2.7 Never Say Never .............................................. 113
  5.2.8 Implications for Maintenance ................................ 114

Jens Andexer  September 26, 2001
5.3 A Hierarchy of Secrets .............................................. 115
  5.3.1 Relate Programs in a Uses Hierarchy ................. 116
  5.3.2 Align the Invocation Hierarchy with the Uses Hierarchy .... 116
  5.3.3 Simplify Downward ....................................... 117
  5.3.4 Use to Simplify .......................................... 118
  5.3.5 Implications for Maintenance .......................... 118
5.4 Designing in the Presence of Undesired Events ............... 119
  5.4.1 Providing for a Program's Responses to Detected UEs ..... 120
  5.4.2 Provide Sufficiency in UE detection .................. 120
  5.4.3 Providing for Human Responses to Detected UEs ....... 122
5.5 Guidelines for Maintenance-in-the-Large ...................... 124
  5.5.1 Programming Standards ................................. 124
  5.5.2 Promote close Adherence to Programming Standards .... 125
  5.5.3 Avoid Very Small and Very Large? ..................... 126
  5.5.4 Promote Single Entry, Single Exit ..................... 126
  5.5.5 Use White Space Consistently .......................... 127
  5.5.6 Use Beacons .......................................... 129
  5.5.7 Keep Naming Conventions Simple ...................... 129
  5.5.8 Organize Programs According to Uses Structure ....... 130
  5.5.9 Generalize instead of Clone ........................... 130
  5.5.10 Provide a Test Environment that is as Close to Production as Possible ..................................... 131
  5.5.11 Minimise the Depth of Inheritance ................... 131
  5.5.12 Implications for Maintenance ......................... 131
5.6 Conclusion .................................................. 132

6 Documentation Paradigms ........................................ 135
  6.1 The Role of Documentation .................................. 136
    6.1.1 Subsetable .......................................... 137
    6.1.2 Precise ........................................... 137
    6.1.3 Complete ................................ .......... 138
    6.1.4 Current ........................................... 138
    6.1.5 Verifiable ................................ .......... 138
    6.1.6 Standardized ....................................... 139
    6.1.7 Many Audiences, One Set of Documents .............. 142
  6.2 Sources of Documentation ................................... 143
    6.2.1 Pre-Development ..................................... 143
    6.2.2 Development ......................................... 144

Jens Andexer  September 26, 2001
6.2.3 Maintenance ................................................. 156
6.3 Conclusion ..................................................... 163

7 Case Study ......................................................... 167
  7.1 Key Points in the Case Study ................................. 168
    7.1.1 Reuse in Documentation ................................. 169
    7.1.2 Simplicity ............................................... 169
    7.1.3 Standards .............................................. 170
    7.1.4 The Index .............................................. 171
    7.1.5 Cross Referencing Information ......................... 172
  7.2 How the Documentation Supports Maintenance .............. 174
    7.2.1 Adaptive and Perfective Maintenance ................... 174
    7.2.2 Corrective Maintenance .................................. 176
  7.3 Observations Made during the case study .................... 178
    7.3.1 Document formatting ................................... 178
    7.3.2 Document Structure that is Fixed by the Medium ....... 178
    7.3.3 Inadequacies of Documentation Media .................. 179
  7.4 Conclusion .................................................. 180

8 Conclusion ....................................................... 181
  8.1 Applications for This Work ................................ 181
  8.2 Limitations ............................................... 182
  8.3 Future Work .............................................. 182
  8.4 Conclusion ................................................ 184

9 Standards used in this Thesis ................................ 187
  9.1 Notation .................................................. 187
  9.2 Index Notation ............................................ 187
  9.3 Glossary of Acronyms ...................................... 188
  9.4 Glossary of Terms .......................................... 190

Thesis Index ..................................................... 210
  Thesis Index .................................................. 211

Bibliography ...................................................... 222

A Software Risks ............................................... 243

B Published Estimates of Maintenance Costs ..................... 247

Jens Andexer  September 26, 2001
| CONTENTS |
|------------------|---|
| C  Case Study: Ace Insurance | 249 |
| Ace HOF Index | 1 |
List of Tables

2.1 Time-to-fix in C and C++ .................................................. 32
3.1 Maintenance Activities and Effort Spent ............................... 39
4.1 Sizes of Methods in Three Object Oriented Systems ............... 70
B.1 Published Estimates of Maintenance Costs ........................... 247
List of Figures

2.1 Conversation Between a Researcher and a “Typical” Software Maintainer 11
2.2 Java Classes ...................................................... 24
2.3 Published Estimates of Maintenance Costs .......................... 33

4.1 Software Change from Domain Need to Production ............... 42
4.2 Domain Maps ...................................................... 54

5.1 Base Machine ...................................................... 109
5.2 Virtual Machine ...................................................... 110
Chapter 1

Introduction

For over thirty years the digital computer has proven itself to be a versatile machine. The popular and professional press, full of new computer applications, are testimony to this technology’s prolific growth, a growth built on flexibility.

No success story, however, is without its problems; nagging at the heels of the digital computer’s flexibility is cost. Software maintenance specifically poses the costliest challenge faced by the computer industry.

Computers are attractive because they are an inexpensive and versatile solution to many real-world problems. Indeed, it is the computer’s ability to be applied to a wide range of problems that is its most attractive feature. But what is it about computers that makes these machines so easily adapted to solving new problems?

The computer’s adaptability to new problems comes from its modifiable behaviour. A computer program describes a computer’s behaviour making programs the key to its versatility.

Although the program specifies a computer’s behaviour, the program itself has no moving parts. In this way, computer programs that give the computer its versatility can never wear out an - idea that seems too good to be true. The software that makes the computer a cheap and versatile solution to real-world problems is also

Jens Andexer

September 26, 2001
maintenance free. But, as Parnas points out in [130], not wearing out does not make software maintenance free.

The trouble is that the real-world changes, and in order for computer systems to stay useful they must be adapted to this changed world. In fact, changes in the real-world are non-stop, and to stay abreast of the real-world many computer systems must undergo almost continuous change.

All of this change, however, comes at a price. As observed in other branches of science all systems that undergo continuous change characteristically tend toward disorder\(^1\)[41]. Unfortunately disorder makes software more difficult and more costly to modify while increasing the likelihood of introducing new errors\[^{39}\]. It is, therefore, the disorder in the software that makes the use of computers to solve real-world problems more expensive.

Recently, new hope for slowing software’s deterioration into disorder has come from the Object Oriented (OO) paradigm. The Object Oriented paradigm provides a model for a method of analysis that examines requirements from the perspective of classes of objects found in the vocabulary of the real-world. [12][57][102] argue that Object Oriented programs are faster to develop and take less effort to maintain than non Object Oriented programs.

Unfortunately the Object Oriented paradigm too is not without problems. When Object Oriented programs are poorly designed the result can be worse than in non-Object Oriented technologies [31] [30] [144] [175]. As Schneidewind observes in [150]

“Unless we design for maintenance, we will always be in a lot of trouble after systems go into production.”

\(^1\)A tragic example of this is the Apollo 12 disaster, where, as the result of frequent changes to wiring a short circuit caused a fire which claimed the lives of three astronauts. Although the design of the wiring was good to start with, its many changes caused the wiring to degrade into a “rats nest”. [6]

\(^2\)This term is defined in the glossary.
This raises the question “How can systems in general, and Object Oriented systems in particular, be designed to minimise maintenance effort”?

1.1 Purpose

The event that separates development from maintenance is the delivery\(^3\) of the software product\(^4\) to its sponsor. Before delivery the work carried out on software is referred to as development. After delivery the software is put into production where it is relied upon by its users. The work done on software that is in production is referred to as maintenance.

In much the same way that a doctor inherits the responsibility for treating the health problems of a new patient’s misspent youth, software maintainers inherit problems that have their origin in software development. The patient, full of youthful exuberance, may have engaged unknowingly in activities that have detrimental effects in later life. The problem is simply that the patient, in this case the programmer, does not understand the effect today’s actions will have in later years.

The development of software is far from well understood. The impact that design decisions have on the final product are often difficult to assess. Maintenance is so far removed from software design that the impact design decisions have on the maintainability of the final product are difficult to foresee.

The difficulties associated with assessing the impact of design decisions on maintainability are exacerbated when new technologies such as those in the Object Oriented paradigm are introduced. New technologies add another variable to an already uncertain environment.

\(^3\)Software delivery is the point in time when the software sponsor accepts the software from the developer. Delivery can be a complicated process because it is usually part of the fulfillment of a contractual agreement. Delivery can, however, also be extremely informal, but at its core is the acceptance of the software by its sponsor.

\(^4\)This term is defined in the glossary.
1.2 Scope

This thesis removes some of the uncertainty from designing software for maintainability by providing answers to the question “How can systems in general, and Object Oriented systems in particular, be designed for easier maintenance?” This is done by analysing the software maintenance process in order to isolate that which makes the software difficult to maintain. Once the difficulties have been identified points that can be addressed by software design and maintenance will be isolated; from these the principles of design for maintainability will be developed.

1.3 Related Work

The Object Oriented paradigm’s implications for maintenance are not yet fully understood. Personal experience has been reported in [175], [176] and [29], while programmer input was sought in [31], [30] as well as through interviews and interactions with programmers.

Design guidelines for Object Oriented programs, while not discussed in detail, are presented in [102] [103] and [11]. Documentation of Object Oriented programs is also discussed in general terms in each of these works.

1.4 Outline

In order to answer the large question of what makes software more maintainable some basic questions are addressed. First, what is software maintenance? Second, what factors contribute to making software maintenance difficult? Third, how can the factors that make software maintenance difficult be addressed in the software design? Fourth, what documentation is needed to adequately describe the design?

When these questions have been resolved the answers will be applied to a case
study. Here the advantages of designing and documenting software for maintainability are presented in a practical application. The thesis is structured as follows:

Chapter 2 describes the effects of time on software;

Chapter 3 introduces the categories of software maintenance;

Chapter 4 introduces the software maintenance process and describes the difficulties encountered during software maintenance;

Chapter 5 introduces guidelines that, when followed, will minimise the effects of the problems identified in Chapter 4;

Chapter 6 introduces documentation methods which will improve the documentation of Object Oriented systems;

Chapter 7 is a case study of a system developed under the Object Oriented paradigm to determine where it violates the principles of design for maintainability.
Chapter 2

Software and The Effects of Time

Delivery to its sponsor represents a turning point for software. It is during the period after delivery, while the software is under maintenance, that software is subjected most significantly to the effects of time.

The changes made to software from the time it enters production are collectively referred to as the software’s maintenance history. This chapter describes the dynamics at work in software during its maintenance history and how programmers and owners can be affected by these dynamics. Here the Object Oriented paradigm is introduced as a possible response to the effects described.

2.1 Maintenance-in-the-Large

In [34] DeRemer and Kron make the distinction between Programming-in-the-Small and Programming-in-the-Large by observing that

...structuring a large collection of modules into a “system” is an essentially different intellectual activity from constructing the individual modules.

Jens Andexer

September 26, 2001
Software maintenance can be generally referred to as the post implementation work done on software although the I.E.E.E. tends the following more precise definition:

**Definition 2.1 (Software Maintenance)**

“Modification of a software product\(^2\) after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment” [71].

It should be noted that both the general description and the definition provide no information about the software that goes into maintenance.

Parnas, in [44], describes *software engineering* as designing useful programs under one or both of the following conditions:

1. More than one person is involved in the development or the use of the software.

2. More than one version of the software will be produced.

It is the programs produced through software engineering that become software under maintenance. Therefore, in addition to the I.E.E.E’s definition, software maintenance can be described as software modification under one or more of the following conditions:

1. More than one person is involved in modifying the software.

2. One or more people rely on the software.

3. More than one version of the software exists.

\(^2\)DeRemer and Kron refer to a *module* in [34] as “...a segment of [LPS] code”. For this thesis module is defined in Section 5.1.

\(^3\)This term is defined in the glossary.
2.1 Maintenance-in-the-Large

The conditions under which software is maintained cause some of the problems for maintainers. These problems are described in the following sections.

2.1.1 Several Maintainers

Software maintenance is most often done by groups of people working on systems of interacting programs. A change to a program in one part of the system can have an effect on the work being done on another, completely unrelated, part. Unfortunately someone else may be working on that part of the system at the same time. When code is reused throughout a system and the reused code is undergoing modification conflicts between maintainers can occur, and hinder the maintenance effort.

2.1.2 By-products of Change

One by-product of maintenance is that many versions of the software become available. The versions may be grouped and introduced into production as a release or they may be introduced into production on an ad-hoc basis.

Problems arise when different versions of software interact. For example, the rating mechanism for an insurance product may require modification as part of a re-positioning of the product in the market, to take place in five months and three days time. These modifications may be done by one maintenance group working on one release of the software. During the product re-positioning modifications, however, a change in tax legislation requires all products to be taxed at a higher rate. The new tax is to come into effect in four months five days time at 00:00:01 hours. Because the changes resulting from the new legislation affect all products and must come into effect at a very specific time these are put into a release of their own. The problem here is that both dates are fixed and there is not enough time before the introduction of the tax change to fit and test the software for the firm’s own product re-positioning

Jens Andexer

September 26, 2001
changes. The two groups are therefore in conflict, a common situation in software maintenance.

A second by-product of software maintenance is that it is very likely that programs will be modified more than once. One study found that ten generations of maintenance programmers work on the same program before it is rewritten[165].

2.1.3 Risk

Software that has been deployed by an organisation, and is relied on to do business, is said to be in production software. Because production software is often relied upon by a large number of people, or because it is critical for the operation of the organisation, its failures take on special significance. Failures that affect a critical part of the organisation’s daily operation have consequences that can be catastrophic (For details please refer to Appendix A).

2.1.4 Deadlines

Tight deadlines often force maintainers to move modifications into production quickly (see Figure 2.1 on page 11). The significance of Figure 2.1 is twofold. First, it shows the need of the maintainer to identify what must be modified, to make the modifications, and to verify the modifications as quickly as possible.4 Second, the pressures in the maintenance environment hasten the effects of software deterioration5 because software is often modified with an incomplete understanding of the impact of the change. This accelerates the effects of the “Rule of increasing Entropy” resulting in a situation where software that is heavily relied on by an organisation also degrades more quickly because of the pressures of responding to user needs.

4Furugama in [53] found that mental stress brought about by schedule pressure, work load and physical conditions contributed to increasing the faults introduced into software by up to 50%.

5The meaning of deterioration is described later in this chapter.
Figure 2.1: Conversation Between a Researcher and a “Typical” Software Maintainer

**Maintainer** What kind of research do you do these days?

**Researcher** I am initiating some work on program understanding.

**Maintainer** What is that?

**Researcher** Well ... you recover the design of a software system from the source code so you know how it does whatever it does.

**Maintainer** You mean the design of the whole system?

**Researcher** Yes.

**Maintainer** Interesting. Why would you want to do that?

**Researcher** So that you can understand the system when you are modifying it.

**Maintainer** Hmmmm. I can’t see how it would help me in my work.

**Researcher** Well then, what would you like?

**Maintainer** I’d like something that helps me get the job done, fast.

**Researcher** Which is?

**Maintainer** Help me find what needs to be changed, make the change, and get the hell out.

**Researcher** Well wouldn’t the design of the system help them*. 

**Maintainer** It will, but you don’t need the design of the whole system, only the part that needs to be changed.

From Lakhotia in [85].

*Programmers

### 2.2 Software under Maintenance

Circumstances arise which compel software owners to undertake modification. This section analyses the consequences of modifying software and some factors that accel-
erate their ill effects.

2.2.1 The consequences of modifying software

Once the decision to make modifications has been taken the most compelling question is “What effect will the modifications have in addition to changing the software’s behaviour?”.

2.2.1.1 Software Gets Bigger During Maintenance

Most changes made to software involve the extension of functionality\(^7\), requiring the addition of code. Even when the change could be accommodated by rearranging existing code the option to add new code will usually be more attractive to programmers. The attraction is that the impact of new code is easier to assess than the impact of modifying it. It is therefore safer for programmers to add code than it is to change existing code; the net effect being that the software tends to get bigger during maintenance.

To describe software as getting bigger, however, is meaningless without a conceptually simple ordinal\(^8\) measure of size. For this thesis the size of software will be measured as the lines of executable code or LXC\(^9\).

2.2.1.2 Software Gets More Complex During Maintenance

Connections in software can be referred to as dependencies. A software dependency is a relationship in the software such that a programmer modifying one part of the software must be concerned with possible side effects in other parts of the software[175].

---

\(^6\)This term is defined in the glossary.

\(^7\)See perfective maintenance in Table 3.1 on page 39.

\(^8\)An ordinal scale of measure allows a ranking of objects according to some ordering criterion[15]. This is minimally necessary to be able to make a meaningful comparison between two objects.

\(^9\)This term is defined in the glossary.
The following are basic types of dependencies:

**data dependency**: a relationship between two variables, like \( a = a + b \).

**control dependency**: a relationship between two programs, like an invocation.

**functional dependency**: a relationship between a program and its variables.

**definitional dependency**: a relationship between a variable and its type.

Lehman and Belady use *complexity* as an inverse measure of “the degree to which the program is intelligible” [86, Pg. 254]; the more complex a program is the less intelligible it is. They go on to point out that the “intelligibility” of a computer program depends primarily on the “extent and pattern of program interconnectivity; on program structure”\(^\text{10}\).

This allows complexity to be defined:

**Definition 2.2 (Software Complexity)**

* A measure of complexity is the number and the pattern of dependencies that exist in the software.

Complex software, using this definition, has either a large number of dependencies making the software difficult to understand or it has a pattern of dependencies that makes the software difficult to understand or both.

New code adds dependencies to the software that make it more complex. This trend toward greater software complexity was observed by Lehman and Belady and described as follows:

“Rule of Increasing Complexity”

“As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.” *M. M. Lehman and L. A. Belady* [86, Pg. 253]
2.2.2 Software Deteriorates During Maintenance

The trend toward greater size and complexity of software during its maintenance history does not, by itself, constitute deterioration. These are only symptoms of deeper problems in the software. In this case to be able to treat the symptoms one must understand the disease and to understand the disease one must understand what it attacks.

2.2.2.1 What are Program Structures?

A “software structure refers to a partial description of the software showing it as collections of parts and showing the relationship between the parts” [122]. A hierarchical structure has no “loops” in the relationships between the parts[124].

Parnas observes in [124] that with these definitions it is possible to divide any system into parts and contrive a relation that will provide a hierarchical structure. Therefore, to make a description of software deterioration useful, the structure or structures being referred to must be identified.

In this thesis the relation “uses” will be referred to frequently as a relationship between parts of the software. Program A is said to “use” Program B when the correct\(^\text{11}\) execution\(^\text{12}\) of B may be necessary for A to complete its work\([127]\)^\(^\text{13}\).

2.2.2.2 What is Software Deterioration?

Parnas observes in [128] that a deterioration in program structures\(^\text{14}\) is a set of structures that are worse after a modification than the structures that would have resulted

\(^{11}\text{This term is defined in the glossary.}\)
\(^{12}\text{This term is defined in the glossary.}\)
\(^{13}\text{Notice that the uses structure is not the same as the invocation structure. Program A may call program B without relying on B’s correct execution. A more detailed explanation with examples can be found in Section 5.3.2.}\)

Jens Andexer  
September 26, 2001
had the new functionality been included in the original design.

One form of software deterioration is a deterioration in the software’s structures, structures that are laid down during the software’s original design. Symptomatic of deterioration is an increase in the number of connections in the software.

The “Rule of Increasing Complexity” predicts the deterioration of software as modifications are made. This presents the question, “Why does this deterioration happen?”

### 2.2.2.3 Why Does Software Deteriorate?

How a system is divided into parts and the relationships between the parts are described in the software design. One of the most important products of the software design process is a description of the software’s structure.

Often software designers will have a simple idea, or design concept, which helps them resolve design issues uniformly. The uniformity offered by the design concept simplifies the software design[130].

When the original design concept is either not understood or ignored while making changes the software’s structure will almost always degrade[130]. Systems that frequently undergo such change characteristically tend toward disorder[41]. The really bad news about this process is that, as the software degrades over time, the deterioration of software structures accelerates as is described in the ‘rule of increasing entropy’.

“Rule of increasing Entropy”

“The entropy of a system increases with time unless specific work is executed to maintain or reduce it.” [86, Pg150]

---

14Structures would be more accurate according to [124] since software has several structures like the invocation structure and the “uses” structure described later in the thesis.
2.2.3 How Does Software Deterioration Affect Programmers?

The effects of software deterioration on programmers are described by Schneidewind in [150] as follows:

- “It becomes more and more difficult to determine if changing the code will have an effect in another part of the system”;

- “It becomes more difficult to relate features that the software provides with the code that provides them.”

Therefore, as software degrades it takes more effort to modify. This relationship between software and the effort to maintain it allows software maintainability to be defined as follows:

**Definition 2.3 (Maintainability)**

“The ease with which a software system can be modified in order to change or add capabilities, improve performance or correct defects.”[100]

2.2.4 How Does Software Deterioration Affect its Owner?

Owners of software are affected by its degradation in four significant ways.

First, as software deteriorates, it takes more effort to make changes to it (because of its increased size and complexity). The increased effort to make changes to the software translates into increased personnel costs.

Second, software deterioration means that its owners cannot respond as quickly to changes in the real world, which, in turn affects the organization’s competitiveness [130].

Third, a hallmark of software deterioration is unwanted behaviour. As the software’s size and complexity increases, the software becomes increasingly more difficult
to modify without introducing new errors as well as becoming increasingly impractical to test. As a result it is usually not tested adequately and flawed programs are included in the production system. [130]  

Fourth, a casualty of software deterioration is performance. As the software gets bigger it requires more space [130]; this adversely affects performance.

The relationship between the software’s deterioration and the pursuant costs to its owner is described in Lehman and Balady’s third rule.

“Rule of Continuing Change”

“A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost-effective to replace the system with a recreated15 version.”

[86, Pg150]

2.3 Is Object Orientation a Remedy for Software Maintenance Woes?

Programs solve real-world problems by specifying a computer’s behaviour. The computer’s behaviour, in turn, models some part of the real-world. For example, an inventory system retains data about the status of articles under its control. ’Nails’ are a likely article about which an inventory system in a hardware store would have data - the data might be each nail’s size and how many pounds of loose nails of that description are available in the store. But the available Kilograms of a certain size of nail is unlikely to be correct due to inaccuracies in bulk weight. Therefore, the inventory system can only give an idea of how many kilograms of a certain nail

15The version resulting from a new development.
are available. The data in the inventory system only approximates, or models the real-world, it is not the real-world nor is it typically an exact representation.

The Object Oriented paradigm is based on the proposition that the more closely a program’s structure resembles the real-world problem it is to solve, the better the program will be\textsuperscript{16}[100]. This section describes how the Object Oriented paradigm models the real-world.

2.3.1 The real-world

Since the Object Oriented paradigm is an analogy of the real-world this section will describe the real-world characteristics important in the Object Oriented paradigm.

2.3.1.1 Objects

Let us assume that the reader is capable of identifying objects in the real-world. If this is so, then one of the objects the reader might identify is him or her self.

2.3.1.2 Classification

Upon reflection, readers might also identify some categories to which they belong, for example, the reader will very likely belong to the category called 'humans’. Humans are in a category whose members all share some common characteristics, characteristics that suffice to classify the reader as being an object of the type human and not an object of the type 'chair’ or 'dog’ or ‘gorilla’ or ‘chimpanzee’.

2.3.1.3 Properties

Beyond the characteristics that allow the reader to be classified as human are some additional properties that help differentiate the reader from other members in the same

\textsuperscript{16}McConnell does not define what is meant by better. But for the purposes of this thesis maintainability will be assumed to be one of the qualities of “better”.

Jens Andexer

September 26, 2001
category. These differentiating properties such as 'eye colour', 'hair colour'\textsuperscript{17}, 'vocation' and others provide additional data about individuals in real-world categories. This additional data is important because it allows the category members to be individualised.\textsuperscript{18}

There are two kinds of individualising properties. One kind is unchanging or static, the other is changing or dynamic. Static properties are those where the data is the same for all members of the category. The data for dynamic properties may differ between members of the category but may also change for the same member over time. For example, all humans have one head and the potential for hair on their head\textsuperscript{19}. These properties are static, they will not change during the course of a human's life time. The extent to which that hair is present or absent and its colour can, however, differ between individuals and may change in the same individual over time. These properties are dynamic.

2.3.1.4 Behaviour

In addition to the characteristics and properties described above, objects have behaviour. How a real-world object's behaviour is specified is beyond the scope of this thesis. Let us assume, however, that a human's possible behaviours are to be retained somehow in that object. Therefore, new behaviour must add or modify the behaviour already retained in the object.

If a person wanted to learn to play an instrument, say a saxophone, this behaviour would add to an already existing pallet of behaviour. Notice that if another object already knows how to play the saxophone then both objects would be able to play the

\textsuperscript{17} Assuming the reader has hair. If not then consider it the colour it used to be.
\textsuperscript{18} Although properties supply additional data nothing prevents them from also contributing to the classification process.
\textsuperscript{19} The astute reader will notice there are exceptions to even this rule but let it be assumed for this thesis that all humans have exactly one head and that all who have one also possess the potential for hair thereon.
saxophone but the playing might be different. Both objects would have “saxophone playing” behaviour but it would differ between the objects.

2.3.1.5 Identity

Although classification and properties help differentiate one object from another they do not always allow an object to be uniquely identified and are certainly not a convenient identification mechanism. To make identification easier many objects are provided with an identity; in humans the most commonly used identifier is a name. In general, an objects identity allows it, and things associated with it, to be addressed without confusion. But names are not necessarily unique, sometimes they are only unique in conjunction with other properties or within a certain context. In the real-world characteristics must be added to an object to differentiate one from another. For example, if there are two people called Jeff, one might be differentiated from the other through additional properties: “Jeff with curly hair”, or through limiting the context by saying “Jeff at school”.

2.3.1.6 Relationships

The developers of the Object Oriented paradigm favoured sentient real-world objects, like humans, when they developed a model for inter object relationships. Perhaps the two most important of these are hereditary relationships and friendships. Through the inheritance relationship a person can be loaded with characteristics that come from any one of a long line of descendants. Friends, conversely, build associations that do not load either member in the relationship with characteristics of another member. In a friendship the relationship does not burden or benefit the participants with heredity.

If Jeff had a friend, Bill, then Bill could help Jeff learn the saxophone. In this relationship Jeff would not be burdened by Bill’s genes. Jeff’s hair would not straighten
because Bill has straight hair while he is helping Jeff learn to play saxophone. No friend will benefit from characteristics like talent\(^{20}\) that come from Jeff’s genes. Jeff can only benefit from his descendant’s genes and only Jeff’s descendants will benefit from Jeff’s genes. Therefore, if Jeff has a great talent for playing saxophone then he cannot thank Bill for this, Jeff’s ancestors are responsible\(^{21}\).

### 2.3.1.7 Communication

In the real-world objects often need to communicate with one another, how else could Bill influence Jeff’s playing other than to signal Jeff so that he can exhibit certain behaviours. This communication, in the form of a message, must have a sender and at least one receiver although its medium can be any of those used during human interaction. The sender is clear, it is the originator of the message. The receiver’s identity is not clear. In a general broadcast all objects within the range of the broadcast can receive the message. In point to point communication there is exactly one sender of the message and one receiver. In order for Jeff and Bill to communicate point to point every message would need to be addressed only to each other.

### 2.3.2 The programming-world

This section will describe how each real-world characteristic introduced above is implemented in the Object Oriented programming languages and where there are important differences between the real-world and its implementation in the paradigm.

#### 2.3.2.1 Objects

An idea central to the Object Oriented paradigm is that objects in the programming world should model objects in the real-world. The model, however, will be imperfect

\(^{20}\)It is assumed for the sake of this argument that talent is hereditary.

\(^{21}\)Random chance in the form of mutations also plays a role but it is assumed to be unimportant since mutation plays no role in the inheritance relation on the Object Oriented paradigm.
because of imprecision that originate from two sources. First, there are limitations on how the real world can be measured as well as limits on how precisely real-world data can be stored. Second, not all the properties of real-world objects can be stored due to cost.\textsuperscript{22} Therefore, only sufficient data about the real-world will be stored to make the program an acceptably good\textsuperscript{23} model.

In the real-world objects typically exist to begin with and based on the characteristics of their existence real-world objects can be classified into categories. In the programming world, however, objects are created through a process called \textit{instantiation} and destroyed through a process called \textit{destruction} during an Object Oriented program’s execution\textsuperscript{24}. In the programming world the category always comes first; the category’s definition describing the data and behaviour of its member objects.

To make defining categories and their relationships easier Object Oriented programming languages provide special programming constructs. The following sections describe some important parts of Object Oriented programs and the programming constructs used to make their implementation easier.

\textbf{2.3.2.2 Classification}

Classification is the process of categorising objects of the same type. In the Object Oriented paradigm a \textit{type} refers to the data and behaviour that is associated with an object and the type is implemented in a programming construct called a \textit{class}(\textit{For details please refer to} Figure 2.2 A). Arguably the most important programming construct offered by Object Oriented programming languages, the class serves as the template from which objects are created through instanciation (\textbf{F} in Figure 2.2). As the process’ name implies an object is an instance of a class it has the data and

\textsuperscript{22}An example of this is the Y2K problem described in Section 4.4.1.2.
\textsuperscript{23}What is acceptably good is a question of costs versus benefits that is beyond the scope of this thesis.
\textsuperscript{24}This term is defined in the glossary.
2.3 Is Object Orientation a Remedy for Software Maintenance Woes?

behaviour defined in the class, it has the class’s type. All objects instanciated from
the same class are of the same type.  

2.3.2.3 Properties

Just as in the real-world objects in the programming world have properties that
provide additional data about the object. This data is important to individualising
objects so that they better represent the real-world object they model. Program
properties are implemented as variables\(^{26}\), Q in Figure 2.2, that define an object’s
state, or data state. Booch defines state as follows:

Definition 2.4 ( State )

“The state of an object encompasses all of the properties of the object plus the current
values of each of these properties.” [12]

Variables can be static or dynamic. Static variables, I in Figure 2.2, are also
called class variables and are allocated once per class. Each object, or instance of the
class, then shares the same class variables. The variable maxSides, in Figure 2.2,
is allocated once for all instances of polygon and once for all instances of rectangle
and, therefore, in this model all polygons have a maximum of 100 sides and all
rectangles have a maximum of four sides. Although imperfect, polygons can have
an infinite number of sides in the real-world; this limitation will be close enough for
this thesis. It is important to note, though, that the key word ’static’ does not mean
that maxSides cannot be changed. It would be possible to code maxSide = 5; in

\(^{25}\)Booch points out in [12], page 59, that type and class are not exactly the same because classes
also define interclass relationships. Booch goes on to observe that separating the concepts is confusing
and of little value. Booch’s stance is acceptable for the arguments tendered in this thesis.

\(^{26}\)This term is defined in the glossary.
Figure 2.2: Java Classes
2.3 Is Object Orientation a Remedy for Software Maintenance Woes?

the Rectangle and, as soon as the code is executed, all rectangle objects instanciated using the Rectangle class would have a maximum of five sides. In the programming-world ‘static’ does not mean unchanging, it really means common to all instances of a class.

Dynamic variables are also called instance variables (Q in Figure 2.2). All variables are instance variables unless they have the static key word and contain data associated to one object alone, the object of which they are a part. Instance variables are the most common variables used to store data about dynamic real-world properties.

2.3.2.4 Behaviour

Not unlike in the real-world, programming objects have behaviour. Here too an object’s behaviour consists of the actions it can perform. In Object Oriented programs the actions an object can perform are specified in methods, E in Figure 2.2. Unlike the real-world where a person can learn an instrument programming objects typically do not spontaneously learn new behaviour\textsuperscript{27}. For a programming object to change its possible behaviours the new behaviour must be programmed in the classes methods first.

Two important events in an object’s existence are its creation and its destruction. To be called into existence an object must be instanciated. Often instanciation requires the help of a special method called a constructor, G in Figure 2.2. Constructor methods provide a means for performing all the functions like initialising variables required to bring about the object’s initial state.

Once instanciated an object’s data can be manipulated, each manipulation changing the object’s state. The object will persist, or stay in existence, until it is destroyed. An object’s destruction, like its instanciation, is often helped by a special method,

\textsuperscript{27}The astute reader will argue that such learning is possible. But this type of behaviour modification is outside the scope of this thesis.

Jens Andexer

September 26, 2001
the destructor\textsuperscript{28}. The destructor method ensures the object has given up control of all the resources it used. An object’s destruction is either initiated explicitly or automatically when there are no more references to the object. The automated function is called \textit{garbage collection} and runs periodically to destroy unneeded objects.

In the description of the real-world it was noted that two objects can have behaviour within an identity but that the behaviour itself may be different. This phenomenon is called \textit{polymorphism}, meaning multiple behaviours, in the Object Oriented paradigm. Implicit here is that the multiple behaviours are associated with the same method name. Since this requires the introduction of identities polymorphism will be described in more detail in the next section.

\subsection*{2.3.2.5 Identity}

In Object Oriented programming languages objects are referred to with an identifier or a name. In fact, even parts smaller than objects, like methods or variables, must be referenced. Unambiguous referencing of program parts is of central importance in all programming languages but, as will be described in this section, Object Oriented programming languages do not always allow unambiguous references using only names.

The mechanism used to associate an identifier with a program part is called a \textit{declaration}. Declaration does more than associate only a name with a program part. The declaration of a variable, \texttt{Q} in Figure 2.2, associates a variable name with its type and may also allocate memory for the variable. The declaration of a method, \texttt{E} in Figure 2.2, defines the method’s behaviour. The declaration for a class, \texttt{B} in Figure 2.2, defines the data and behaviour the objects instanciated using that class will have. This is where new types can be defined.

A declaration is not always valid in all parts of a program. The part of the program  

\footnotesize{\textsuperscript{28}Also called the finalize method[18]}
where a declaration is valid is referred to as its \textit{scope}. When the declaration is valid throughout the program it is said to be global\textsuperscript{29} but most declarations are restricted to a context or some smaller part of the program. This is one way of using the same name more than once in the same program. Just as in the real-world the name is unambiguous within a context, within its scope.

Typically names must always be unique within their scope. In Object Oriented programming languages, however, methods do not need unique names. Method names can be \textit{overloaded}, meaning the same name can be used by more than one method within the same scope. The correct method can be identified for an invocation by first using the name of the object to which the method belongs, then by using the method’s name and finally by using the method’s \textit{signature}. The signature is the pattern of parameter types along with the type of the returned data. This is the effect called \textit{polymorphism} and is not unlike both Jeff and Bill being able to play the saxophone but their playing being different. The behaviour’s name is the same, 'playing saxophone', but the implementation of the behaviour is different.

\subsection{2.3.2.6 Relationships}

As in the real-world, objects in the programming world have relationships with each other. Here again the two most important relationships are those of \textit{inheritance} and friendship.

Inheritance, \textit{J} and \textit{L} in Figure 2.2, is a relationship between classes that allows one class, the parent, to share properties and behaviour with one or more, even all, its descendants. The class from which another class inherits is called the \textit{super-class} (\textit{K} in Figure 2.2) while the class doing the inheriting is called the \textit{sub-class}.

\textsuperscript{29}The most common of these is the declaration of global variables. These are variables that can be changed from anywhere in the program. Although these are referred to as global variables, in fact it is the scope of the declaration that makes the variable global

Jens Andexer  

September 26, 2001
Two varieties of inheritance are possible: *single inheritance* and *multiple inheritance*. Single inheritance means that a child, sub-class, can have exactly one parent, the super-class. The sub-class still inherits from all the ancestors of the super-class but there can be only one parent. Multiple inheritance means a sub-class can inherit from greater than one super-class. In this case the sub-class inherits from all the ancestors of its super-classes.

The inheritance relationship makes it possible to access data and methods implemented in any ancestor using the part's name alone. For example method $setSide_m$ implemented in $Polygon_e$ can be invoked from $Square_e$ in the same way it is invoked in $Rectangle_e$. The inheritance tree is searched back through the descendants until a match is found, no matter how far back. The only limitation is through access designators which are explained later in this section.

Friendship in the real-world is a relationship in which one participant can rely on another. In the Object Oriented paradigm this type of relationship is called *containment* and is shown as P in Figure 2.2. In a containment relation one class uses another as the type for a variable. This allows the class using the type to rely on the services of the implementing class without sharing its properties. Unlike inheritance, variables and methods can only be shared between objects participating in the containment relation because in order to refer to a variable or a method the objects name must be used. For example, P in Figure 2.2 shows the declaration of the class $Side_e$. $Side_e$ is used as a type for the array $sides_v$ in $Polygon_e$. Notice, though, that the methods implemented in $Side_e$ can only be referenced using $side_v$. In order to use the method $putLength_m$ the following statement must be coded:

$$sides[pv\_side].putLength(pv\_length)$$

Here the $putLength_m$ method associated with an object of type $Side_e$ at position $pv\_side$ in the array of objects called $side_v$ is invoked.
2.3 Is Object Orientation a Remedy for Software Maintenance Woes?

Class relationships allow variables and methods to be shared by changing their scope. A declared part can be seen on one of the two sides of a class, on the inside or on the outside. The potential scope of a declared part is determined by the access designator, D in Figure 2.2. The following are the three most commonly used access designations:

**public** Declarations defined as public are visible inside and outside the declaring class and inside all its descendants. Methods and data declared in this way can also be seen through the containment relation (D in Figure 2.2).

**protected** Declarations defined as protected are visible only inside the declaring class and inside its direct descendant, or inside its children. Parts designated as protected can also be seen through the containment relation (D in Figure 2.2).

**private** Declarations defined as private are visible only inside the declaring class. These parts cannot be seen through inheritance or through containment (D in Figure 2.2).

### 2.3.2.7 Communication

In the Object Oriented programming world messages are always point-to-point. The sending object uses the receiver’s identity to direct the message whose purpose it is to initiate the execution of a method in order to take advantage of its behaviour. The initiated method can be part of the same class description or part of a different class description, making it possible to initiate methods in other objects. The benefit is that the behaviour need not be implemented in the sending object. In Figure 2.2, M shows a message whose sender is Square, and whose receiver is the setWidth method in Rectangle (N in Figure 2.2). The message itself is the parameter pv_side, shown in 0.
2.3.2.8 Beyond Reality

So far the real-world has been used as a analogy for Object Oriented concepts. Unfortunately the analogy breaks down with some of the more esoteric concepts in the Object Oriented paradigm. This section will describe these subtler concepts.

2.3.2.8.1 Overriding

Overriding is an extreme form of polymorphism. In the type of polymorphism described above method names are overloaded and the ambiguity in naming is resolved using the method signatures. It is, however, possible to declare methods with the same name and the same signature, although they must be in different classes. When the classes that declare methods with the same name and the same signature are related through inheritance the method declared in the sub-class overrides the method in the super-class. That means when a message is sent to an object with overridden methods the methods in the sub-class will get the message, not the method in the super-class.

2.3.2.8.2 Abstract Parts

Sometimes it is convenient to define classes that have data and methods useful to a related set of sub-classes. A class called \texttt{GraphicalObject}, could be defined to serve as a super-class for \texttt{Polygon} as well as \texttt{Circle}\textsuperscript{30}. This makes \texttt{GraphicalObject} a convenient place to implement methods and data common to all graphical objects. An object with a type of \texttt{GraphicalObject} is confusing because it is not clear which category of graphical objects is meant. For this reason Object Oriented programming languages allow for abstract classes.

Abstract classes cannot be instanciated, they can only be used as a super-class in an inheritance relation. They are a programming convenience that allows related data and methods to be defined once and re-used without the risk of the super-class becoming an object.

\textsuperscript{30}This example is from [18].
2.3 Is Object Orientation a Remedy for Software Maintenance Woes?

The abstract class can force sub-classes to re-implement some of its methods. This is done by designating the method as abstract. An abstract method contains no program code, it is only the declaration of the method name along with its signature. Abstract methods must be implemented, however, in a sub-class of the class where the abstract method is declared. Therefore, the implementation of the method overrides the abstract declaration of the method.

2.3.3 Is OO Better for Maintenance?

Clearly the Object Oriented paradigm is a divergence from traditional process oriented thinking. Just because Object Orientation is new, however, does not make it a good idea. One must ask if making program structures resemble real-world structures is a good idea. Do Object Oriented programs make software easier, and therefore less costly, to maintain?

Table 2.1 provides data revealing that it takes longer to correct errors in C++ than in C, a trend continued in [70], while [21] observes that classes involved in inheritance contained six times more errors than those not using inheritance. Although worrying, these statistics do not allow the conclusion that Object Oriented software is necessarily less maintainable. The statistics do, however, suggest that it is possible to introduce serious maintenance difficulties when developing Object Oriented systems.

There are no easy answers to the questions above. The remainder of this thesis analyses what makes Object Oriented programs difficult to maintain and what makes them easy to maintain in order to derive software development guidelines that will help the maintainer.
Table 2.1: Time-to-fix in C and C++

<table>
<thead>
<tr>
<th>Language</th>
<th>Total Hours</th>
<th>Total Defects</th>
<th>Hours / Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>375.1</td>
<td>74</td>
<td>5.5</td>
</tr>
<tr>
<td>C++</td>
<td>1341.0</td>
<td>94</td>
<td>14.3</td>
</tr>
</tbody>
</table>

2.4 Conclusion

Figure 2.3 presents, in graphical form, published estimates of maintenance costs since 1965 that are presented, in detail, in Appendix B. The estimates vary between approximately 30% to 80% of an organisation’s data processing budget depending on how the data was collected and what cost was included in their calculation. Although there is considerable variation in this data, it shows a slight upward trend. Due, however, to variability in the data and its collection predictive value is minimal. However, the data should suffice to conclude that maintenance costs are not falling.

The depressing stubbornness of maintenance costs makes it appear that a costly post-delivery period is inevitable. This is not necessarily the case. Boon makes the following observation in [13]:

“The main problem here is that maintenance can be made an intolerably difficult task if the preceding design approach ignored testing and maintenance needs.”

If this is true then the causes of software deterioration have their origins in the software’s original design [88][150][128]. If software was not originally designed to allow for changes later on, then modifying the software will inevitably lead to its

\[31\text{For details please refer to Table B}\]
2.4 Conclusion

Figure 2.3: Published Estimates of Maintenance Costs

![Graph showing published estimates of maintenance costs over time.]

Please see Appendix B for sources.

deterioration [121]. Preventing software deterioration requires an understanding of the process used to change it. To this end the software maintenance process must be understood in greater detail. The following chapters study the software maintenance process and analyse what makes it difficult so that design guidelines can be developed to make a difficult process less so.
Chapter 3

Software Maintenance Categories

In their study of 487 data processing organisations, Lientz and Swanson, [93], identified three categories of software maintenance. These categories represent the three primary motivations for software maintenance which have become accepted as the industry norm for categorizing maintenance activities. This section describes these three software maintenance categories and concludes with an analysis of how much effort is spent in each category.

3.1 Corrective Maintenance

Software is correct if, and only if, it meets its specification. Whether or not a program meets its specification can only be proven through mathematical verification [125].

A software error occurs when the software displays unexpected behaviour; colloquially called having a “bug”. The unexpected behaviour can originate in the specification or it can be the result of the software not being correct. Corrective maintenance entails the diagnosis and correction of errors in the software after the software has been delivered, regardless of their origins.
3.1.1 Errors that Originate in Software Requirements Specification

Software requirements are gathered by analysing the proposed software’s interaction with other software and hardware but, regardless of this, requirements are usually gathered from people. Unfortunately people are an imperfect source of information.

Problems begin when those people who commission the building of software do not know exactly what they want, as is typically the case. Even if what is desirable is known the requirements are usually not understood at the level of detail needed for software development. Through careful analysis the needed details can be added but this comes at a cost. Gathering and analysing requirements is a process during which an organisation learns a great deal about itself and how it does business. This learning process very often results in changes to the requirements as the gulf between what was thought to be true and what actually is true closes. Gathering and analysing requirements also takes time during which external influences, like new legislation, can force changes to the requirements. The results of these problems create ambiguity. Different people will understand the same problem differently because the problem is changing and because their understanding of it is changing. This misunderstanding often leads to errors in the requirements.

Errors in the software requirements specification can manifest themselves in two ways. The software may behave correctly according to the requirements but the requirements specify unacceptable behaviour. Alternatively, the requirements may not specify desired behaviour and the software, consequently, does not fulfil that requirement.\(^1\)

\(^1\)Notice that these “bugs” and errors have their origins in poor requirements and not in poor programming.
3.1.2 Errors that Originate in Implementation

There are two categories of errors that originate in software implementation.

First, the program may violate its specified behaviour. In this case the program is not correct and must be modified to behave as specified. Second, even when the program is correct an event can occur that is not desired and requires special processing. How is this special processing initiated?

Consider adding 1 to a 1 byte long unsigned integer variable that already has all bits set to binary 1. The operation will result in an incident called an ‘overflow’ because the value resulting from the add operation is too large to be stored in the 1 byte variable. The overflow triggers an event called the ‘overflow event’, but to initiate special processing the event must be detected. Detection allows some special processing to take place that will attempt to handle the overflow event. If the event handling processing cannot return the program to a state where normal case processing can proceed then the event must be reported to the program that was executing when the overflow was detected. If the invoking program also cannot continue its normal case processing as a result of the detected event, this program too may report the error to its invoker in a process called propagation.

In this thesis events needing special processing are referred to as “undesired events” (UEs [142]) and fall into two categories:

1. incidents - incidents are UEs which, although undesired, were expected and for which recovery is successful. These are not errors, because they are part of the software’s specified behaviour and because the specified behaviour is the desired behaviour. An example is a screen input field validation that checks if user entered data is numeric. If the data is not numeric the software detects this and displays an error message. In the overflow example above, if the program in which the overflow is detected handles the error, by setting the variables value
to zero, then the overflow is an incident.

2. *crashes* - errors for which recovery is unsuccessful or for which no recovery was attempted. An example of a crash is a software *failure* that causes the program to end abnormally. An abnormal end, also called *ABEND* for ABnormal END, means a severe error causes special error handling routines to be invoked. An overflow can result in a crash. If the software does not detect the overflow before the arithmetic operation (or if no recovery from this error is made) the program will *fail.*

### 3.2 Adaptive Maintenance

During adaptive maintenance the software is modified to accommodate changes in its technical hardware and software environment.

### 3.3 Perfective Maintenance

Perfective maintenance involves enhancing the system to add new capabilities. These new capabilities include new user requirements and enhancements made in the interest of performance or efficiency.

### 3.4 Allocation of Maintenance Effort

Maintenance effort allocated among the three maintenance categories has been found to be distributed according to Table 3.1. Table 3.1 shows how significant Perfective Maintenance is compared to the other categories.

Although the categories are mutually exclusive a software modification may belong to more than one category. For example, a correction may require the redesign
of a module, improving its design, and thus fall into corrective and preventative maintenance. Similarly, changes made as a part of any category of maintenance activities always risk introducing new errors which necessitates increasing the effort dedicated to corrective maintenance.

Table 3.1: Maintenance Activities and Effort Spent

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrective Maintenance</td>
<td>21%</td>
</tr>
<tr>
<td>Adaptive Maintenance</td>
<td>25%</td>
</tr>
<tr>
<td>Perceptive Maintenance</td>
<td>50%²</td>
</tr>
<tr>
<td>Preventative Maintenance³</td>
<td>4%</td>
</tr>
</tbody>
</table>

²A survey in [93] of 486 managers of large commercial data processing centres indicated their greatest concern in maintenance is the demand for software enhancement.

³This term is described in the glossary but because this category of maintenance only accounts for a small percentage of effort it will not be included further in this thesis.
Chapter 4

A Software Maintenance Process

A *software maintenance process* is the series of steps a programmer performs when making software modifications. Although there are several processes, the so called “quick fix” maintenance process[48], predominates in industry. For that reason, it will be discussed in this thesis.

The Quick Fix process consists of the following steps:

1. Defining and understanding the change.

2. Reviewing the documentation.

3. Comprehending the programs.

4. Implementating the program modifications.

5. Testing the modified programs.

6. Updating the documentation.

Although the process appears to be a simple sequence of steps maintainers usually cycle through the steps a number of times. The most significant cycling, shown graphically in Figure 4.1, is typically from testing where errors are found back to

Jens Andexer

September 26, 2001
comprehension where the reason for the error must be determined. The process ends when the software passes an acceptance test and is installed for users. The time that elapses between detecting an error and its correction is referred to as the “time-to-fix” the error.

Figure 4.1: Software Change from Domain Need to Production

This diagram was developed by the author.

This chapter consists of sections that correspond to steps in the quick fix maintenance process for software. Each section contains a description of the step and an analysis of features that make work during that step difficult. The analysis will be based on formal academic papers, experience reports, and informal information gathered from maintenance programmers in the field.
4.1 Defining and Understanding the Change

Most software changes are initiated by users working in the software’s area of application. The application area for which a software system is written is referred to as the *problem domain*.

During perfective maintenance the software maintenance process is initiated by a *change request* or CR. Particularly in perfective maintenance the CR is the means of communicating changes in the business, the problem domain, to maintainers. For example, property insurance is a problem domain for which insurance software is written. The knowledge built up about the property insurance business is the maintainers’ *problem domain knowledge*.

A CR may also initiate adaptive maintenance. When an application must be adapted to accommodate new hardware or new software these changes can also be described in a CR.

During corrective maintenance the process is usually not initiated by a CR. Depending on the urgency of the needed correction defining the change may be dispensed with or severely curtailed in order to speed up the process. This is true in particular when the correction is in response to a crash that results in program failure. Program failures, by their nature, identify the clear requirement: to recover from the failure and to correct its cause.

4.1.1 Difficulties

Problems associated with the gathering and understanding of user requirements abound and almost always result in *requirements volatility*. This is a situation where requirements change after work on later steps in the software maintenance process has started. Because the work in later steps relies on earlier work, this phenomenon often results in wasted time and effort. The following sections identify some of the most
important causes of requirements volatility.

4.1.1.1 User uncertainty

Users are familiar with daily activities in their operational areas. But it is the nature of adaptive maintenance to exist outside the daily routine. Changes in the business environment can be complex and can have far reaching effects that make them difficult to understand.

4.1.1.2 Inconsistencies

A business process is a set of steps that must be performed to complete a task, for example, giving a quote for home owner insurance. Programming a business process into a computer system requires a very detailed understanding of each step in that process. This need for a detailed understanding typically necessitates a close inspection of how the organisation does its business.

Usually organisations rely on business processes once they are in place and only make the effort to have them analysed when changes are needed. This periodic analysis often uncovers inconsistencies in how different users understand the same business process and inconsistencies in how the same business process is performed by different parts of the same organisation\(^1\). These inconsistencies are another source of requirements volatility.

4.1.1.3 Different Cultures

Technical staff understand the inner workings of software but are usually less familiar with the part of the organisation it supports. Users operate in a very different world from programmers. Often it is difficult for users to understand what it is possible for the software to deliver, or to understand what the final product will look like, let alone

\(^1\)Thanks to [9] for informative discussions on this point.
how it will benefit them. Conversely, maintainers often lack some of the background problem domain knowledge users take for granted. The result of the difference in cultures is inaccuracies, omissions and ambiguities in the CR.

4.1.2 Observations

The greatest obstacle to understanding the change request is the lack of precision in the original requirement. The CR describes the requirement for a change, the lack of precision in the CR is symptomatic of an incomplete understanding of the problem. Change requests are inaccurate because users must describe a change they do not yet fully understand to maintainers who do not have the same understanding of the user domain. This leads to misunderstandings, as was described in Section 3.1.1, often resulting in errors in the requirements.

The CR is a tool for users to document their understanding of changes in the problem domain. When CRs are written precisely the inaccuracies and ambiguities identified above can be identified before work starts. Precision in the CR is helped by a detailed description of the current system\(^2\). Understanding what the system does at the outset provides a basis upon which new requirements can be added.

4.2 Document Review

The analysis of any large software product should begin with a review of its documentation. During perfective maintenance, documentation can serve as a description of what the software’s behaviour is. This description would then serve as a base onto which new functionality can be added by developing specifications for the new behaviour.

During corrective maintenance documentation could be consulted to determine

\(^2\)For details please refer to Section 6.2.2.1.
how UEs are handled by the software. Crashes would usually not be documented as part of the software’s description since these are often events outside the software's specified behaviour. Past crashes would, however, have been recorded and the corrective action logged. These logs could be reviewed to see if the error has occurred before and what action was taken if it had \(^3\) [61].

Documentation could also help during adaptive maintenance in describing how the software interacts with the technical environment. Where the technical environment changes the documentation could be consulted to identify where the software must be modified to accommodate the technical change.

### 4.2.1 Difficulties

Unfortunately documentation does not often deliver the above information. Maintainers are plagued by poor supporting documentation [23]. The following problems are encountered in software documentation, including comments from the code text.

#### 4.2.1.1 Unavailable Documentation

Often software is viewed only as a means of delivering functionality. In this view the delivery of functionality is the primary objective and activities that are not part of the programming effort are considered unproductive. Documentation is, therefore, seen as overhead and will not be produced or kept current unless ample resources, people and money are available. As resources become scarce, user requirements are turned into software functionality with less effort spent on intermediate steps like requirements documentation\(^4\). This situation becomes acute as deadlines loom and the user requirements change.

---

\(^3\)Singer identifies information about past failures as key to maintainers engaged in corrective maintenance in [156].

\(^4\)A lack of effort to produce supporting documentation is identified as a problem by Dart \textit{et al.} in [32] and is identified as a problem in maintenance by Singer in [156].
There are many reasons why documentation may not be available to maintainers. One of the most frequent reasons is that the documentation is simply lost, a victim of neglect.

4.2.1.2 Incomplete Documentation

As described above, documentation is not the primary objective of software development projects. Sometimes it is produced at the end of the development process after the programs have been written. Since the priority of the developers is to deliver an operational software product, documentation can be left incomplete. The developer’s primary objective is a complete set of functionality [32].

4.2.1.3 Inaccurate Documentation

Most maintainers view software documentation with some trepidation because when it is available it is inaccurate. Documentation can be inaccurate when it is created or can become inaccurate by being incorrectly updated or by not being up-dated at all. For detailed descriptions of software behaviour inaccurate documentation is worthless [156]. If the documentation is not maintained when the software undergoes changes it will become inconsistent with the production software and will no longer be trusted or used.

4.2.1.4 Imprecise Documentation

Frequently documentation written to describe a computer system is written imprecisely using natural language. Natural language is a poor tool for communicating precise details about program behaviour. As a result programmers often find it easier to read the code text [136].
4.2.1.5 Poor Indexation

Documentation that is written in natural language has the tendency to be written like a story. Individual documents are written to be read from front to back. Searching through this type of documentation can become a greater navigational challenge than searching through the source code text [156].

4.2.1.6 Redundancy

When the documentation is poorly structured it is difficult to tell what has been included and where it can be found. With more than one person working on the documentation this problem becomes acute and redundancy is often the result.

4.2.1.7 Inaccessible Information

When deadlines are close and resources short it often seems better to produce something rather than nothing at all. Documentation that is produced under these circumstances has the information the developer thought was important when the program was written. But under stress the information is often presented in a way that is simply confusing.

Multinational organisations often create multinational software projects where the requirements produced in one country are shipped overseas to be implemented or where people with different languages must work together. These circumstances invariably mean that documentation must be translated from one language to another. Even with the best efforts of translators there will be loss information during the translation.

\footnote{Parnas also makes this observation in [136].}
4.2.2 Observations

The fundamental problem identified in most of the points above is the lack of formality in documentation, in the structure of sets of documents to support a software system, in the structure of individual documents and in the language used to communicate technical details. Singer [156], reports most maintainers agree the 'code is king'. It is the final definitive source of information about the software's behaviour. The reason the 'code is king' is that the documentation is unreliable or difficult to use.

Singer makes another important observation in [156] in that programmers will only consult the code when they cannot find someone who is familiar with the program. Another programmer is often a more effective source of information than source code text and for documentation to be used it too must be a more effective source of information than the source code text.\footnote{This observation is verified by the author's own experience and by [61].}

In order to be a more effective source of information than other sources\footnote{For details please refer to Section 4.3.1.} documentation must summarize the source code precisely. If the documentation is not an accessible summary of what the programs do it will not be used. If the documentation is not precise it also will also not be used.

4.3 Program Comprehension

Dijkstra, in [36], makes the observation that a program exists in two forms, as static text and as a dynamic process. A program's static text describes the set of instructions that specify its control logic. A program's dynamic process results from the execution of the program. During a program's execution the control logic specifies the order in which program statements are executed.\footnote{Notice that Dijkstra makes the assumption that programs are not self modifying. Self modifying programs can change parts of their control logic while they execute. The static code text for these}

\footnote{This observation is verified by the author's own experience and by [61].}
\footnote{For details please refer to Section 4.3.1.}
\footnote{Notice that Dijkstra makes the assumption that programs are not self modifying. Self modifying programs can change parts of their control logic while they execute. The static code text for these}
When the program is small enough and the control logic is simple enough a programmer can inspect the static text and visualize the dynamic process directly \[108\].

That is, in Floyd’s \[49\] terms, the programmer can understand the meaning of a program, see what the program does with data\(^9\).

Software systems are neither small nor are they typically simple \(^{10}\). In order to understand what programs do with data programmers must have ways to manage the details with which they are confronted while still trying to tease information from the source code.

One way programmers manage detail is to forget it, an approach called “Just in Time Comprehension”\[158\]. Singer et al. observe in \[158\] that maintainers are content to understand just enough of the program to make the change required and to confirm to themselves that the change they have made will have the desired effect\(^{11}\). After working in one particular area of the software programmers’ quickly forget details when they move to another part of the system \[158\]. The benefit of this strategy is that the programmer’s knowledge is constantly refreshed with the most current information about the software. This fresh information comes at the expense of putting a continuous load on the sources of information available to programmers, sources that can result in misinterpretation and mistakes.

The following section begins with a description of the sources of information available to maintainers. It goes on to analyse how this information is used and what makes the information difficult to obtain. Finally, observations will be made about how information can be more effectively extracted from one of its sources, the software.

---

\(^9\)Floyd in \[49\] does not provide a dynamic view of computer programs. Floyd provides a basis for the formal definitions of the meanings of programs so that rigorous standards are established for proofs about computer programs.

\(^{10}\)See Figure 2.3 on page 33.

\(^{11}\)This can also be seen in Figure 2.1.
4.3 Program Comprehension

4.3.1 Sources of Information

When maintainers try to understand software they do so with a clear objective: to fix a particular “bug” or to implement a particular enhancement. Questions that maintainers want to answer are similar to the following:

“How is adding a risk to an existing policy implemented?”

or

“Why does the data base not get updated when the sum assured is non-zero after the decimal point?”

Here the sources of information used for program comprehension during software maintenance are influenced by the category of software maintenance. With different objectives for the comprehension process it follows that the way information is extracted from the sources of information will also vary.

This section describes the sources of information available to maintainers, how information is extracted from each source and how the extracted information is used.

4.3.1.1 Documentation

The role documentation plays in program understanding was described in detail in Section 4.2.

4.3.1.2 Users

Users can provide a great deal of information about the software’s observable behaviour. However, this information must be judged for its technical content in the light of the programmers’ experience. This is particularly the case during perfective maintenance when requirements are unclear and when users have a poor understanding of what is needed. During corrective maintenance too, users provide valuable
information about the behaviour that took place during a UE. It would probably be a user who notices that non-zeros after the decimal in the sum assured lead to the database not being updated. This information can be tempered with frustration by the user’s inability to perform their work.

4.3.1.3 Problem Domain Knowledge

Knowledge tends to be organised according to observable characteristics[169], providing a “user’s” view of the software’s functionality. A plan is a special type of knowledge that supports the development and the validation of expectations about how one domain maps onto another[169]. A plan can be thought of as knowledge consisting of a problem and expectations about how the problem is solved, expectations gained through experience. Von Mayrhauser et al. observe in [5] that knowledge is usually organised into plans and goes on to observe in [169] that plans tend to reside in long term memory.

A domain plan provides an understanding about how the problem domain translates into software behaviour. Therefore, if domain knowledge is an understanding about how the fire insurance cover is written as part of a home owner’s insurance policy a domain plan would be an understanding of how writing the fire insurance cover for a home owner policy is done using the software system being studied. The domain plan maps the knowledge about writing a policy onto the knowledge about how to use the screens in a software system.

In Example 4.3.1, problem domain knowledge might be how to “add a risk to the existing fire insurance cover” and a domain plan might be knowing how to write that cover using a software system. Looking at Figure 4.2, this type of knowledge would be retained in mapping A. In mapping A, the knowledge of how a risk is added to a policy is translated from knowledge in terms of the domain:
1. Add a policy page to the original reference, 001345 in this case, by crossing out the pre-printed reference and writing the original reference.

2. Fill in the risk information for the additional risk.

3. Add the new risk to the original policy folder.

This is done using the software system by:

1. Use screen G1HF325 to add Risk address.

2. Use screen G1HF326 to add Risk details.\textsuperscript{12}

\textbf{Example 4.3.1 (Problem Domain Knowledge)}

Insurance protects the insured from the financial loss resulting from a risk. The risk is what is insured, like a building, but it is insured against a peril, like damage from fire or damage from flood. A hazard is a specific situation that increases the probability of a loss arising from a peril actually occurring, or that may influence the extent of the loss. A wood shingle roof or a long distance to the nearest fire hydrant are hazards because the wood shingle roof increases the probability of a loss due to fire while the distance to the fire hydrant influences the extent of the potential loss. The coverage provided by an insurance policy describes the protection provided under a contract of insurance while the cover is the contract of insurance. Therefore it is possible to have several risks under the same cover.\textsuperscript{[35]}

An application of the above knowledge could be writing a home owner’s insurance policy. This policy can have a number of different types of coverage ranging from home contents to fire insurance for the building. The cover for fire insurance might cover the primary residence but could be extended to cover a summer cottage located kilometres away on a lake, two risks under the same cover. The cottage on the lake, recently purchased, is a wood frame construction built in 1963 with a tar shingle roof located 500 meters from the nearest fire hydrant.

\textsuperscript{12}Notice that capturing the risk information delocalises over more than one data input screen.
Figure 4.2: Domain Maps

Note: code fragments are in Java

Note: Problem domain tasks, like writing fire insurance cover for a risk, map onto screens through domain plans (mapping A). The screens map to the implementation through programming domain plans (mapping B).
4.3 Program Comprehension

4.3.1.3.1 How Problem Domain Knowledge is Used

Knowledge about the problem domain supplies a context that helps in understanding a software system. The context makes it easier to understand insurance software, for example, when the maintainer has a good understanding of the insurance business that the software was written for. This knowledge helps guide the analysis of a problem and helps focus attention on what is important. This is knowledge gained through experience and it is knowledge that allows maintainers to more effectively decompose and comprehend problems[169].

Problem domain knowledge is independent of the software implementation. A business process is the same business process regardless of whether it is implemented in a procedural language, like COBOL\textsuperscript{TM} or PL/I\textsuperscript{TM}, or in an Object Oriented language like Java\textsuperscript{TM}. The process of writing a new insurance policy for house insurance must result in the same premium regardless of the programming language the premium rating\textsuperscript{13} software is written in. The mechanics of how the policy is written may, however, differ. A progressive company may offer its agents the ability to write policies over a widely expanded network. A conservative company may require the policy to be filled out on preprinted forms and mailed to a central data entry location. The business of creating a policy based on a financial product offered by the insurer remains unchanged in both cases.

Knowledge about the problem domain is clearly important during perfective maintenance. Here, experienced maintainers use their knowledge of the user domain to identify functionality that will be affected by new or changed requirements. Knowing how “adding a risk to an existing policy is implemented” guides the maintainer in identifying what parts of the implementation need to be changed. This allows the maintainers to use top-down analysis. R. Brooks in [17] describes the top-down com-

\textsuperscript{13}Rating is part of the process of assessing how much a customer must pay for an insurance policy.
prehension model as mapping knowledge about the user domain to source code. To identify the source code the maintainers must identify the behaviour or the functionality that addresses the user requirements. First, therefore, the requirement must be understood in the context of the user domain; in the insurance example the maintainer must first understand what is involved in the business of writing a policy and how the software supports this process. Second, and described in the next section, a maintainer must understand how screens used to write a policy are implemented in code. Therefore, the mapping R. Brooks identifies is built up beginning from a two step mapping, shown in Figure 4.2, and ending with the programmers own mapping scenario.

During corrective maintenance and adaptive maintenance knowledge of the problem domain helps identify the users and the parts of the system affected by a problem or by a technical change.

4.3.1.4 Technical Knowledge

Understanding how user requirements are translated into functionality is the first step in a two step process. The second step is understanding how the functionality is implemented in code, knowledge that is remembered using plans.

Soloway et al. in [161] describe programming plans as “program fragments that represent stereotypic action sequences in program code and the mechanism for retaining characteristic code patterns in long term memory”. An example of such code patterns is the addition of a new risk, shown in Example 4.3.2. Prior experience of how this type of behaviour is implemented drives expectations about how the behaviour will be implemented in the software being studied. This allows the programmer to more effectively identify key pieces of code that need more detailed attention.
Example 4.3.2 (Technical Domain Knowledge)

Knowledge of the technical domain begins with understanding the syntax of the programming language, in this case Java\textsuperscript{TM}, and extends to knowledge about what is implemented in the programming language. Knowledge of the implementation might be understanding the classes used to implement risk in the application.

**Code Implementation in: Java**

```java
public class RiskPortfolio {
    Risk[] portfolio;
    int  riskCount;
    int  maxRisks;
}

public void addRisk ( Risk risk ) {
    // make sure the risk id only occurs once in the portfolio
    if ( searchForRisk( risk.riskId ) < 999 ) {
        riskCount ++;
        portfolio[ riskCount ] = risk;
    }
}

public void deleteRisk( Risk r ) {
    int i = searchForRisk( r.riskId );
    portfolio[ i ].removeRisk();
    return;
}

public int searchForRisk( int riskId ) {
    for (int i = 0; i<= maxRisks; i++) {
        if ( portfolio[ i ].getRiskId() == riskId ) {
            if ( portfolio[ i ].getDeleted() == false ) {
                return  i;
            }
        }
    }
    return 999;
}
```

*This example is continued in Example 4.3.3 on page 58*
Example 4.3.3 (Technical Domain Knowledge, continued)

**Code Implementation in: Java**

```java
public class Risk {
    int riskId;
    String[] riskAddress;
    int riskInsurdValue;
    int riskBuilding;
    boolean riskDeleted;
}

public Risk (int id) {
    riskId = id;
    riskAddress = new String[100];
    riskInsurdValue = 0;
    riskDeleted = false;
}

public void removeRisk() {
    riskDeleted = true;
}

Note: in order to save space many details, like dates, important to an insurance application have been omitted.
```

4.3.1.4.1 How Technical Knowledge is Used  During perfective maintenance technical knowledge allows programmers to concentrate less on how functionality is implemented and more on where it is implemented in the code text, as can be seen in Example 4.3.4. Because programming plans link implementation to functionality when these plans exist they must only be triggered to allow this abstraction to be used. Singer *et al*. in [158] observe that inexperienced maintainers are less focused and will spend more time studying things in the source code that are not relevant
to the problem. The experience a maintenance programmer has in understanding someone else’s code improves their effectiveness and efficiency in code understanding [169] because they have a greater body of technical knowledge to draw from.

**Example 4.3.4 (Program Domain Mapping)**

An example of mapping in the program domain can be seen in Figure 4.2, mapping B. Here the knowledge of how the screens for creating a new risk are implemented in a Java™ code fragment. In mapping B, the knowledge, in terms of screens, is mapped onto code chunks. Here knowledge in terms of

1. Use screen G1HF325e to add Risk address.
2. Use screen G1HF326e to add Risk details.

is mapped onto code that implements the functionality provided by the screens. This mapping is typically built up by using the code as a source of information. A programmer builds hypotheses about how the functionality might be implemented and then uses the code to see if they are true or not. Knowing how the adding of a risk might be implemented as code, by having seen it before in other systems, drives a programmer’s expectations. Therefore, knowing that the risk is part of a portfolio of risks, the programmer would expect to find the method for adding a risk to the portfolio in the `RiskPortfolio` class. (see Example 4.3.2)

\*See Section 4.3.1.5.1 for a description of this term.

The technical knowledge used during perfection maintenance is also useful during corrective maintenance tasks, where interdependencies between software systems are also important. For example, part of a maintainer’s technical knowledge might include that shown in Example 4.3.4 on page 59 and extend to knowing that when adding a new risk the data entered on screen will populate an object in the `Risk` array in the `RiskPortfolio` object. This data populates a large DB2™ data base located on a mainframe computer by being passed along to a transaction called `TR415w`. However, the column in the relational data base that corresponds to the sum assured is defined

---

Jens Andexer  
September 26, 2001
as integer. Adding sums assured with zero behind the decimal allow the zeros to be truncated during the conversion from decimal to integer. When adding a sum assured with non zeros behind the decimal point the data base update program returns a non-zero return code that forces the transaction to be rolled back\textsuperscript{14}. This bad return code is not returned to screen $G1HF325e$. The effect is that sums assured that are non zero after the decimal point do not update the data base.

During adaptive maintenance knowledge of the interdependencies between software systems is used to assess the impact of change to the technical environment. For example, technical knowledge of a compiler can help a maintainer assess what effect a new compiler release will have on the system under maintenance.

Understanding how software behaviour is implemented as code, while not getting lost in the technical details of the implementation, are the hallmarks of experienced maintenance programmers.

### 4.3.1.5 Source Code

To determine what software really does inevitably requires analysing source code. Although source code is not the most efficient source of information and its interpretation is prone to errors it will be the most reliable source of information about program behaviour in the absence of documentation.

Schemes used by programmers to understand source code text during different maintenance activities are not the same. Programmers adapt their technique to the requirements of a task and to their own personal preferences. Despite its individualistic nature, some generalities about how programmers extract information from program code can be made.

Central to all schemes used to understand programs are questions. Programmers read programs to answer questions about software behaviour, about what the pro-

\textsuperscript{14}Rolling back means all data modified by the transaction is returned to its original state.
4.3 Program Comprehension

Program is doing or about how the program does something. To answer the questions programmers build hypotheses and then go about finding information to prove or reject them. Ruven Brooks theorizes in [17] that cognition is driven by hypotheses. Letovsky in [92] refers to hypotheses as conjectures and goes on to identify three types of conjecture used by programmers. Why conjectures hypothesize the reason for some function or design choice. How conjectures hypothesize about the method of accomplishing a program goal. What conjectures hypothesize about what something is, like a class or a method.

4.3.1.5.1 How Source Code is Used Hypotheses are important to cognition because they help to define the direction of further investigation.[17] Generating hypotheses about code and investigating whether they can be accepted or must be rejected is an important facet of code understanding.[5] The acceptance or rejection of an hypotheses is established through inquiry episodes. An inquiry episode consists of reading the code, asking questions, conjecturing answers, searching and answering the question. Here the programmer is trying to build a coherent model about what a fragment of code is doing.

Once finished this understanding forms a “chunk” which makes up part or all of a plan.[90] A chunk is a generalisation about a piece of code text, it is the programmers understanding of that code text. Chunking, therefore, is a way for a programmer to assign meaning to a fragment of code. Assigning meaning in this way allows the programmer to release the detail about the code text and move on to understanding another chunk. Shneiderman in [153] describes a process where programmers form chunks from the source code text and then use chunks and code text to form larger chunks so larger and larger parts of the program can be understood.

How effective a programmer is in the chunking process is likely to be dependent
on their familiarity with program comprehension\textsuperscript{15}. Soloway in [161] defines \textit{rules of programming discourse} as conventions used by programmers to set up their expectations about programs. These expectations are then used for more effective chunking. However, these rules are developed by individual programmers to recognize particular constructs. Therefore, increasing the palette of recognised code constructs, it appears to this author, is technical knowledge that comes with experience.

During perfective maintenance the preferred approach to get information from the code text is to work \textit{“top-down”} from the user domain to functionality and from functionality to program code. The \textit{“top-down”} approach relies heavily on domain and technical knowledge to guide code analysis [5]. During \textit{“top-down”} analysis program code is read on an \textit{as-needed} basis which means only reading parts of the code while trying to identify key code fragments that deserve more attention\textsuperscript{16}.

When a code fragment that is key to proving a hypothesis is identified the programmer will analyse this code using a more systematic approach. During \textit{systematic} program comprehension the code fragment is read sequentially and is usually small enough for the programmer to be able to visualize as a dynamic process. Understanding this small dynamic process, sometimes a fraction of the whole program, allows a hypothesis about the program’s behaviour to be proved or disproved to the programmer’s satisfaction. When the hypothesis is proven correct the programmer should understand something more about the implementation of some behaviour.

During corrective maintenance the following questions about the UE\textsuperscript{17} must be answered:

\textsuperscript{15}This assertion is supported in [5] but a survey of the current literature failed to produce empirical support for the notion that improvements in the efficiency of proving hypotheses is related to a programmer’s experience in program understanding.

\textsuperscript{16}The previous section showed that this is a process that relies on knowledge that is gained by experience.

\textsuperscript{17}UEs are described in Section 3.1.2.
4.3 Program Comprehension

1. How did the UE manifest itself as behaviour?

2. What actions must be taken to recover from the UE?

3. Why did the UE occur?

4. Where was the UE detected?

5. Is it practicable to correct the error causing the UE and if so how?

6. Is it practicable to prevent the error causing the UE and if so how?

The first step during corrective software maintenance is to address the question of what happened. Users are usually the best source of information for this since they are often directly affected by, or are the cause of, the UE.

Armed with information about what happened the programmer must rely heavily on domain and technical knowledge to try to determine how the system was affected. Often UEs, and in particular crashes, can result in a system’s data being left in an unacceptable state where continued processing would result in erroneous output. In on-line systems where users interact with the software through screens usually relatively small amounts of data are accessed during a program’s execution and recovery may be restricted to correcting the data being accessed at the time of the UE. In batch, large amounts of data are manipulated in the absence of user intervention. Here large amounts of data may be accessed by a series of programs in what is called a job stream. Recovering from a UE in a job stream can involve a significant amount of work to get data files and databases back to a state where the job can be started again from the beginning, or restarted at some point after the beginning of the job stream.

Once the impact of the UE has been determined its cause must be found. This almost always involves code analysis and relies very heavily on technical knowledge.
Not all UEs have their origins in the code where they are detected since environmental factors can play a role in a UE’s cause. A program that fails because a request for main storage cannot be satisfied will fail at the point where the storage request is made, at the instanciation of the object for example. The program failure, in this case, may be symptomatic of a problem in the management of storage by another program in a different system. If the program reporting the UE cannot determine if an object’s instanciation will be successful before it is attempted then this UE cannot be corrected by changing the program code being analysed.

UEs that do have their origins in the code are typically caused by program behaviour that put the data into the state causing the UE. The behaviour causing the data state that results in a UE is not necessarily near the code where the UE was detected and reported. This could be a complex interaction of behaviour spread throughout the software, in a network environment the behaviour may not even be on the same computer.

Determining how the data got into the state causing the UE involves first finding where in the code the UE was detected and then where it occurred. This analysis requires both domain and technical knowledge, although technical knowledge is primarily relied upon during what can be a long and difficult process.\(^{18}\) Once the site of the UE has been found in the code text, programmers rely heavily on their technical knowledge to trace backwards, bottom-up, through the code to find where the program data was put into the state causing the UE.

While tracing bottom-up the code is mostly read on an as-needed basis and is only read systematically when a suspect code fragment is found. The process continues until the programmer is comfortable that the root of the problem has been identified.

\(^{18}\)Programmers in [61] observe that a significant part of their time during corrective maintenance is spent analysing the output of a program execution to find the relevant point in the code text where a UE reported and where it occurred.
Once identified, the process of implementing a corrective change can be initiated. The key to the programmer ending the analysis is reproducing the UE. In corrective maintenance reproduction of the problem is vital to its solution [156].

During adaptive maintenance the software must be analysed for what impact a change in the technical environment will have on program behaviour. This analysis tends to start with code to identify where the software interacts with the part of the technical environment that is to change. Then, using bottom-up analysis, the impact on functionality can be determined.

4.3.1.6 Other Programmers

Other programmers can be the most effective source of information about software. Singer observes in [156] that programmers will consult source code before other programmers only when they are familiar with the program. If they are not familiar with the program they will consult another programmer, if possible the author of the code. Unfortunately, with very old systems it is unlikely the author is still available and in very large systems there may not be anyone familiar with the part of the system being studied [61].

4.3.2 Difficulties

Although program comprehension is done slightly differently during work done for each maintenance category there are difficulties that can occur during any maintenance task.

4.3.2.1 Mixing Application and Control Data

Certain data, application data, is used in programs to store characteristics of the problem domain. Data in the Risk class in Example 4.3.5 is typical of this type of data and is used to describe some characteristics about the risk. Other data is
primarily used to regulate the flow of control in a program and is not part of the application data. An example of this type of control data is a return code from a method that reports its successful completion to a caller by returning the location of the sought item. In Example 4.3.2 on page 57 this happens when the \textit{searchForRisk}_m method returns 999_k to indicate the key was not found and to give the location in the \textit{riskPortfolio}_v array. Using the 999_k will work, as expected, until a cover has greater than 998 risks. The risk with a \textit{riskId}_v of 999_k, although found, will precipitate behaviour in \textit{searchForRisk}_m that is indistinguishable based on the return code from the \textit{riskId}_m not being found.

\begin{example}(Control Data vs Application Data)\end{example}
\begin{quote}
In the implementation of Example 4.3.1 on page 53 risks are collected into a \textit{RiskPortfolio}_v. Such an implementation requires the ability to add, delete and search for risks within the portfolio using the \textit{riskId}_v as a key to identify each risk in the portfolio uniquely\(^a\). If \textit{searchForRisk}_m does not find the risk it is looking for, a 999_k is returned to indicate the not found condition avoiding the overhead of creating an error object.
\end{quote}

\(^a\)Note that \textit{addRisk}_m guarantees uniqueness.

### 4.3.2.2 Mixing Exception and Normal Case Code

When programming languages have no special constructs for UE detection and reporting the code charged with this task tends to be interspersed in the normal case code. This has the effect of cluttering the normal case code and making it more difficult to understand. At the same time the UE specific code also gets spread out making this more difficult to understand as well. Consider Example 4.3.6 on page 67 where most of the code is used to validate the format of a postal code while the code to actually update the post code is obscured by the validations.
Example 4.3.6 (Programming without UE Detection Constructs)

The following code checks the validity of a Canadian postal code. If the format is correct the post code entered on screen is used to update the post code portion of an address.

Code Implementation in: Java™

```java
int updatePostCode( screenPostCode CDN sPC ) {
    int rc = 0;
    for (int n=1; n<=AlphaUpperCase.length(); n++){
        if ( sPC.substr(1,1) != AlphaUpperCase.substr(n,1) ) {
            rc = -1;
        }
        if ( sPC.substr(3,1) != AlphaUpperCase.substr(n,1) ) {
            rc = -3;
        }
        if ( sPC.substr(5,1) != AlphaUpperCase.substr(n,1) ) {
            rc = -5;
        }
    }
    if (rc==0) {
        for (int n=1; n<=AlphNumeric.length(); n++){
            if ( sPC.substr(2,1) != AlphNumeric.substr(n,1) ) { 
                rc = -2;
            }
            if ( sPC.substr(4,1) != AlphNumeric.substr(n,1) ) { 
                rc = -4;
            }
            if ( sPC.substr(6,1) != AlphNumeric.substr(n,1) ) { 
                rc = -6;
            }
        }
        if (rc == 0) {
            postCode = sPC;
        }
    }
    return rc;
}
```
4.3.2.3 Delocalised Plans

*Plans*\(^{19}\) are useful descriptions of the mechanism that allows programmers to associate characteristic behaviour with program code. Unfortunately plans do not necessarily correspond to any program structures. Plans are delocalised when code that is conceptually related is physically distributed through a software system [97]. This happens in Object Oriented systems because business processes are seldom local to objects. Business processes usually affect more than one object. Since the object is the primary mechanism for encapsulation supplied by the Object Oriented paradigm[12] this results in the behaviour delocalising in the software.

For example, remembering Example 4.3.1 on page 53, a possible enhancement to that system would be to add a RESTORE function that would change the status of a risk from “deleted” back to “active” and could be implemented using riskDeleted = false to mean “deleted”, otherwise the risk is “active”. If the SEARCH function is used from the RESTORE, it must be changed to find “deleted” records or the restore will not work - as the “deleted” records will not be identified. This example can be found in Example 4.3.7 on page 71.

The effects of adding the RESTORE function are not local to the RESTORE function. To make the change correctly the programmer must gather causal information about the effects the change to searchForRisk\(_m\) has on other methods. Causal information describes interactions in the software that produce the delocalised behaviour. In the example there is a functional dependency between the new RESTORE functionality and the existing SEARCH function through riskDeleted\(_a\).

The delocalisation of behaviour increases the load on a programmers cognitive abilities like attention span, *abstract reasoning*\(^{20}\) and short term memory [157]. Letovsky in [91] found that the most frequent trigger for inquiry episodes were lines of code

\(^{19}\)Plans are described in Section 4.3.1.3.

\(^{20}\)This is described in the following sections.
that had non-local interactions. In order to understand this type of code the programmer must track down information about the non-local interactions. This information usually comes from reading the code text, although it could be available as technical knowledge or from other programmers.

This type of causal information is only supplied by using a systematic comprehension strategy. Programmers using the as-needed strategy study only parts of the code which they think are relevant for their needs. This strategy does not render a good understanding of dependencies within the software. Indeed, if there is a dependency between methods while this strategy is being employed it will very likely go undetected, leading to an error if that dependency is related to the change.

It is observed in [175] and in [176] that functionality is embodied in program code differently in the Object Oriented programs than in these programs written using a procedural model. Often behaviour is implemented in Object Oriented programs using very brief methods that can only pass a message through to another method, doing very little processing themselves. As a result Object Oriented systems tend to be made up of many very small methods (see Table 4.1). Systems with many small sub-programs make it easier for a maintainer to understand each sub-program on its own but make it difficult to understand the system as a whole since functionality is spread or delocalised over many small methods [176] [144]. The as-needed strategy is, therefore, prone to errors in Object Oriented programs where methods are small and interrelated at the same time.

4.3.2.4 Interruptions Affecting Attention

Information that is relevant to a maintainer reading code will help prove or disprove a hypothesis. To find information, the programmer must read the code or use search
programs to find identifiable character strings like variable names\textsuperscript{21}. With the small methods and fragmented behaviour described above this can involve tracing a chain of method invocations through different classes that are defined in separate files in order to identify where the work is really getting done [175][149]. Lakhota in [85] observes that these difficulties are acute when the file and directory structure into which programs are organised do not follow a program structure.

A particular pathology for this problem has been called the “Yo Yo Problem” by Taenzer \textit{et al.} in [164]. Sakkinen in [149] describes it as a situation that results from an \texttt{abstract} method invoking an \texttt{abstract} method in the same object. The search goes up and down the inheritance hierarchy, like a “Yo Yo”, until an implementation is found. This problem becomes particularly pathological with multiple inheritance, the \texttt{exponential yo yo problem}[149].

When relevant information is difficult to find in the code text each enquiry episode (used to locate a new piece of information) can become an odyssey of its own. The

\textsuperscript{21}Using products like GREP in UNIX, FIND in Windows or SUPERC in MVS.
Example 4.3.7 (Delocalised Plan)

Since the insurer in Example 4.3.1 on page 53 does not want to lose customer information once it has been captured, risks are not physically deleted from the system. Instead, the risk that is to be deleted is marked as “deleted”, by setting the riskDeleted_v flag to true, and left in the portfolio. Therefore, searchForRisk_m has been made sensitive to “deleted” risks. Only risks that have the riskDeleted_v flag as false_v are covered by the insurer.

Enhancement A new function to restore a risk to the portfolio would change riskDeleted_v back to false_v. In order to implement this change the restoreRisk_m method must be implemented in both the riskPortfolio_c class and in the Risk_c class. In addition searchForRisk_m must be changed to return all risks. The change to the searchForRisk_m method propagates a design decision. If the risk is to be added but exists in the portfolio marked as deleted then what happens? In this implementation this risk still may not be added.

Code Implementation in: Java™

```java
/* new method for portfolio class */
public void restoreRisk( int riskId ) {
    int i = searchForRisk( riskId );
    if ( i < 999 ) {
        portfolio[ i ].restoreRisk();
    }
    return;
}
/* modified method in riskPortfolio class */
public int searchForRisk( int riskId ) {
    for ( int i = 0; i <= maxRisks; i++ ) {
        if ( portfolio[ i ].getRiskId() == riskId ) {
            return i;
        }
    }
    return 999;
}
/* new method for Risk class */
public void restoreRisk() {
    riskDeleted = false;
    return;
}
```

Jens Andexer

September 26, 2001
programmer must change focus from the primary task of understanding the program fragment to a secondary task of tracking down information. The interruption that results can prove very disruptive to the overall comprehension process [158].

4.3.2.5 Increased need for Abstract Reasoning

Some maintenance tasks require the ability to reason about the effects a change will have on a complex set of dependencies in a program [157]. To understand these relationships the programmer must “understand what the program does with data” [108]. To understand what the program does with data programmers must understand the dynamic process of a program and therefore understand the program’s control dependencies. However, as is observed in [40], the dynamic process of an Object Oriented program is very different from the static text. During execution Object Oriented programs are rapidly changing networks of communicating objects. This makes the dynamic process difficult to conceptualize from the static code, particularly when the objects may be on different computers across a network.

Because the dynamic process of an Object Oriented program can be very different from the static text there is a greater need for programmers to reason abstractly. The reason is simply that the more dependencies there are the more difficult it is to understand the full implications of a change. Not understanding its full implications makes a change prone to errors.

4.3.2.6 Limitations of Short Term Memory

Maintainers must often keep track of a large amount of detail while analysing program code [157]. Miller in [106] identifies a limit to the ability for humans to “process information”. Here he describes a theory on the ability of an observer to process information based on psychological experiments in absolute judgement. The observer is considered the communications channel which takes stimuli as input information
and produces responses as output information. Information was measured in bits where one bit of information is the information needed to make a decision about two equally likely alternatives. Therefore, “if we decide whether a man is less than six feet tall or more than six feet tall and we know the chances are 50-50, then we need one bit of information” [106]. By supplying stimuli and measuring the accuracy of responses the capacity of the channel to process information was ascertained to be seven plus or minus two bits of information \(^{22}\). Therefore, information that can be extracted from the detail of a program is limited by the programmer’s ability to process the details.

4.3.2.7 Concept view of Inheritance

The concept oriented view [177] of inheritance is motivated by the notion that Object Oriented programs should model the relationships found in some part of the “real-world” [102]. A typical example using this view is the classification of species where an ‘elephant’ “is-a” ’mammal’ [45][102] characterises the relationship between the subclass ‘elephant’ and the super-class ‘mammal’. This type of classification can lead to programs that are difficult to understand[177].

**Example 4.3.8 (Not a Square?)**

This square is not square.

**Code Implementation in: Java\(^{TM}\)**

```java
Square s = new Square( 10, 20 );
if (mySquare.isSquare() == false){
   \// mySquare.isSquare( ) will evaluate to false
}
```

\(^{22}\)It can be argued that information about programs does not decompose into bits as Miller describes and that this makes Miller’s conclusions unsafe when applied to maintainers. But maintainers are limited by their ability to reason about programs while doing analysis, and Miller provides compelling reasons for these limitations.

---

Jens Andexer | September 26, 2001
The problem arises because subclasses may not drop anything inherited from the super-class. In Example 4.3.9 on page 75 each \textit{Square}_c object contains two data elements (inherited from \textit{Rectangle}_c), one for each side, where one data element would suffice and, indeed, two can cause problems. All methods in the \textit{Rectangle}_c class are also in the \textit{Square}_c class. This makes the construct specifying a non-square in Example 4.3.8 on page 73 possible.

4.3.2.8 Accumulation of Behaviour

Of the three possibilities for limiting the scope of the data and methods defined in a class, typically only public and private are used. Data or methods designated as public are accessible in all subclasses that inherit from the class where they are defined. Therefore, as the inheritance hierarchy deepens, there is an accumulation of data and of methods in the derived classes. This accumulation means that to understand all possible behaviour for a subclass, all the public methods for all the super-classes must also be understood.

4.3.2.9 Conceptual Inconsistencies

Different conceptual problems in the inheritance structure have also been identified by Dvorak in [41]. Here it was found that there is less agreement about assigning subclasses to super-classes at deeper levels of the inheritance hierarchy. This is due to the lack of a clear definition of concepts before the software is developed. For example, is a square a rectangle or not? In the geometry a square is a rectangle but an object designed to store the characteristics of a square may not be an object that can store the characteristics of other rectangles[120]. However, an object designed to store the characteristics of a square may be used to store data for a circle.

The problem is an extension of the problem of accumulated behaviour described above. The farther down in the hierarchy the more classes there are to consider as a
Example 4.3.9 (Is a Square a Rectangle?)

When square is instanciated, the default constructor for Rectangle, instanciated that object. Square then goes on to set the height and width as 10 and 20 respectively. If the isSquare method is invoked after the instanciation of Square, it will return a false. In this case Square is not square because it inherits data from Rectangle, that is inappropriate for a square as a result of adhering to an “is-a” relationship.

Code Implementation in: Java

```java
Square s = new Square( 10, 20 );
if (mySquare.isSquare() == false){
    // mySquare.isSquare( ) will evaluate to false
}

public class Rectangle {
    int height;
    int width;
}
void setHeight(int h){
    height = h;
}
void setWidth(int w){
    width = w;
}
public boolean isSquare( ) {
    if ( height == width )
        return true;
    else
        return false;
}

public class Square extends Rectangle {
}

public Square ( int side ) {
    setHeight( side );
    setWidth( side );
}
```
super class. Therefore, the probability, in the absence of clear classification rules, of
incorrectly classifying a subclass increases. Also, the farther down in the hierarchy
the more specialised classes become. This means that finding the correct class to add
data or behaviour to, or to inherit from in order to enhance functionality, becomes
increasingly difficult. The increased detail and increased specialisation found in de-
derived classes at lower levels in the inheritance hierarchy increases the cognitive load
placed on maintainers. This increased cognitive load, in turn, decreases a maintainer’s
effectiveness.

4.3.3 Observations

Problems in comprehension primarily have to do with identifying key code fragments
and assigning a meaning to these fragments. This is the process of abstracting the
code into meaningful chunks and then linking the chunks together in an effort to
make sense of the program in order to find sites needing modification. Making sense
of the program allows the programmer to understand enough of the system to make
modifications to it and have a reasonable chance at making all the needed modification
and not introducing new errors.

This section has made one thing abundantly clear, code is a poor medium for
communicating information. It was found that even programmers do not like using
programs as a source of information and do so only as a last resort. Unfortunately
the last resort is often necessary.

4.4 Implementation

During comprehension the sites where changes are needed will be located and during
implementation the modifications are made. There is a significant amount of cycling
between comprehension and implementation since new information is very often un-
covered during code modification. After a period of comprehension a programmer will feel confident enough about the effects of a change to start modifying code. The code modification will always affect existing code and will often require the addition of new code to enhance existing functionality.

4.4.1 Difficulties

Difficulties during the implementation of changes in software, almost without exception, stem from dependencies in the software. The class to class relationships, inheritance and containment, alter the scope\textsuperscript{23} of classes to make the following dependencies, identified by Wilde in [175], possible in Object Oriented programs:

1. Class to Class
   
   (a) Class $C_{1c}$ inherits from $C_{2c}$

2. Class to Method
   
   (a) method $M_{m}$ returns object of class $C_{c}$
   (b) Class $C_{c}$ implements method $M_{m}$
   (c) Class $C_{c}$ inherits method $M_{m}$

3. Class to Message
   
   (a) Class $C_{c}$ has a method that can be invoked by message $m_{m}$

4. Class to Variable
   
   (a) Variable $v_{v}$ is an instance\textsuperscript{23} of class $C_{c}$
   (b) Variable $v_{v}$ is a class variable\textsuperscript{23} of class $C_{c}$

\textsuperscript{23}Scope has been described in .
(c) Variable $v_v$ is an instance variable\textsuperscript{23} of class $C_c$

(d) Variable $v_v$ is defined by class $C_c$

5. Method to Variable

(a) Variable $v_v$ is passed as a parameter to method $M_m$

(b) Variable $v_v$ is a local variable in method $M_m$

(c) Variable $v_v$ is a non-local variable used in method $M_m$

6. Method to Message

(a) Message $m_m$ passes control to method $M_m$

(b) Method $M_m$ sends message\textsuperscript{23} $m_m$

7. Method to Method

(a) Method $M_{1m}$ is invoked by method $M_{2m}$

(b) Method $M_{1m}$ overrides\textsuperscript{23} method $M_{2m}$

The explosion of dependencies in Object Oriented programs precipitates an explosion in the number of different structures in an Object Oriented program since each dependency can be part of a different partial description of the system. This increase in the number of structures creates challenges during the implementation of changes.

4.4.1.1 Ripple Effect

Some software changes are simple in the context of the program where they are made but have a ripple effect through the system. This typically happens when programs

\textsuperscript{23}These terms are described in Section 2.3 .
that transform data execute in a sequence, each receiving data from the previous
program, manipulating it and sending it along to the next program.

In [175] it is observed that often in Object Oriented programs very brief methods
are written that simply “pass through” a message to another method with very little
processing. The passing through of messages in this way connects the methods along
the way through the parameters.

Consider the case where method \( m_{1m} \) in class \( C_1 \) invokes \( m_{2m} \) in class \( C_2 \) which
invokes \( m_{3m} \) in class \( C_3 \) which invokes \( m_{4m} \) in class \( C_4 \). Regardless of whether
inheritance or containment are used, if \( M_{4m} \) needs a change that requires a new
parameter to be passed to it and the data for that parameter originates in \( M_{1m} \),
all the methods en route that pass the message through must be changed to pass
through the new data. Here a series of message to class dependencies bring about a
close relationship between all four methods.

In addition to new parameters a change to any method in the sequence can cause a
change in the behaviour of the invoking methods. When one method invokes another
the invoker relies on the behaviour of the invoked method. When the invoked method
is changed its behaviour is modified; this can effect all the methods that invoke the
changed method, either directly or indirectly. If method \( m_{4m} \) is changed it can effect
\( m_{3m} \) as well as rippling through to effect \( m_{2m}, m_{1m} \) and all other methods that rely
on any of these four methods. The ripples associated with the change to a method
will spread out through the inheritance structure\(^{24} \) and the containment structure\(^{25} \)
of the Object Oriented program. This is because when method \( m_{1m} \) invokes method
\( m_{2m} \) it can do so by inheriting from \( C_2 \) or by using \( C_2 \) as a type for a variable.

\(^{24}\)The program structure that results from the inheritance relation.
\(^{25}\)The program structure that results from the containment relation.
4.4.1.2 Dangerous Assumptions

All software is written under some assumptions[87]. Lehman in [89] ventures to estimate there is an assumption made for every 10 lines of code in a “typical” program. Some assumptions, like using a particular database management system will be made explicitly. Many assumptions, though, are implicit, a consequence of designing an algorithm, using an API, defining an interface, or deciding on how some data will be stored.

Parnas points out in [122] that “the connections between modules are the assumptions the modules make about each other” and goes on to point out that “these connections are much more extensive than the calling sequences and control blocks usually shown in system structure descriptions”.

Many organisations store dates without storing the century, this part of the date is then assumed to be “19” - to avoid wasting memory on redundant “19”s [131]. Without the century part of the date, date calculations that span into the next millennium can, and often do, produce unexpected results. Called the Y2K problem, estimates range that 300 to 600 billion US Dollars were spent world-wide[179] in order to address the consequences of this assumption.

Interfaces are connections between programs. An interface defines a protocol of behaviour[18]. The assumption made each time an interface is used is that the interface will not change. Each time an interface is used the effort needed to change the interface goes up.

Java is a programming language developed to allow application programs to run under different operating systems (OS platform) without needing modifications to accommodate the new platform. This is done by isolating application programs from

---

26Lehman does not explain what a typical program is but this author accepts the assertion that there are many assumptions made during programming.

27This is not to be confused with a Java interface which is described later in this thesis.
the operating system using special purpose methods that provide a set of operating
system functions. The special purpose methods are grouped into what is called an
Application Programming Interface or API.

API's are used extensively to allow Java™ programs to access services provided by
non-Java™ software and pre-written language elements like the java.math\textsuperscript{28} package\textsuperscript{29}. This has made it common practice to use the APIs methods throughout a
software system, assuming the API will always provide a standard interface that will
never change.

Using APIs to define an interface is seductive. The danger is that each time the
methods of an API are used a control dependency between the software using the API
and the API is created. The assumption made when using the API as a standard
interface is that the part of the API relied on in this way will not change. Control
over changing the APIs behaviour, however, may not reside with the programmers
charged with maintaining the software that uses the API.

4.4.1.3 Objects as Parameters

A class was defined as a complex type in Section 2.3.2.2. Since variables all have types
and variables can be used as parameters then it follows that variables can be classes
and that variables that are classes can be used as parameters. This is true, however,
just because it is possible to use complex types as parameters does not make it a
good idea. Complex classes as parameters create extremely complex relationships
between at least three objects (the sender, the receiver and the parameter) at the
same time. This type of complex relationship provides more potential for the ripple
effect described above and can lead to severe performance problems\cite{81}.

\textsuperscript{28}Java.math contains classes that allow the manipulation of long integer and floating point numbers.

\textsuperscript{29}This term is defined in the glossary.
4.4.1.4 Dependent Parts

Often systems are not split into parts that will function separately. The *business logic* of an application is the part of the system that addresses the problem domain. The presentation logic is dedicated to presenting data to the users of the application in the correct order and format through screens. Very often the business logic of an application is mixed into the presentation logic. When this happens the on-line programs that format and present the data also implement business functionality. If an application written in this way must change the technology used to present the on-line data, then the business logic must be considered when altering the presentation logic.

For example, calculating the pure risk\textsuperscript{30} portion of an office premium\textsuperscript{30} requires data that relates to a specific risk. If the data for that risk is captured from only one screen, it is tempting to put the calculation of the pure risk premium into the same program with the logic that captures the data from the screen. By doing so preliminary calculations, like calculating the distance to a fire hydrant for fire insurance on rural properties, can be done as the data is captured. This assumes that the business logic for calculating this type of premium will change when the presentation technology changes. If the presentation technology is changed to, say a GUP\textsuperscript{30} from a 3270 Terminal Emulation\textsuperscript{30}, the presentation logic is likely also to need change. That means the business logic must be separated from the logic that presents data on screen. This separation is likely to prove time consuming and costly.

4.4.1.5 Problems Stemming from Control Dependencies

One of the benefits of the Object Oriented paradigm is software reuse\textsuperscript{31} [102][103]. Unfortunately often software is reused in an ad-hoc manner by inheriting a class that

\textsuperscript{30}This term is defined in the glossary.
\textsuperscript{31}This term is defined in Section 5.6 .
has methods that provide functions needed in another method. When this happens loops in the “uses” relation can result, precipitating a situation where all the methods in a loop rely on all the other methods in the loop.\footnote{Parnas identifies the symptoms of loops in the uses relation in [121].}

4.4.1.6 Some Problems Stemming From Inheritance

Inheritance is a very close relationship between two classes which allows the derived class access to all of the exposed behaviour and data in the base class. It is difficult, however, to determine which classes inherit from a super-class. This can result in unexpected behaviour in derived classes when the base class is changed\cite{30} \cite{84} \cite{175}. An illustration of this can be found in Example 4.4.1 where a seemingly insignificant change to $addAll_m$ results in invalid results.

4.4.1.7 Cloning

*Cloning* is the process of copying a program and changing it to fit a new need. One attraction of this practice is that the new program is really an old, usually trustworthy, program plus a few changes. Typically it takes less effort to test these programs because, arguably, only the newly added code requires extensive testing. This allows the program to be moved to production more quickly than a program written from scratch.

Conceptually, the new problem is usually a new variation of an old problem. The new variation was not thought of or was not needed when the original program was written. Now that a solution is needed a new program is written, often with 80\% or more of the identical, original code. Unfortunately an error in the unchanged code will be in both the original and the clone, and in any other clones made from the original or from the clone. Also, because the program is changed the old code may not behave the way it used to. The dependencies between the original code and
Example 4.4.1 (Fragile Base Class Problem)

Original code (C++)
When the `addAll_m` method is invoked from class `B_c`, it will use `add_m` method in class `B_c`, the copy of the method that is local to class `B_c`. Therefore, when `cardinality_m` is invoked, it returns the correct count for `B_c`, the value in variable `B.b`.

Modified Code (C++)
Now when the new `addAll()` method is invoked from class `B_c` it no longer uses `add_m`. Therefore, when `cardinality_m` is invoked from class `B_c` it still returns the value in `B.b` but `B.b` no longer has the correct value since the `add()` method was not invoked to update it. In this example a call to `add_m` is not the same as using `a++_m` because in some cases `add_m` implements `b++_m`.

Implemented in : C++

<table>
<thead>
<tr>
<th>Original Class A</th>
<th>Original Class B</th>
<th>Modified Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class A{</code></td>
<td><code>class B : public A {</code></td>
<td><code>class A{</code></td>
</tr>
<tr>
<td><code>public:</code></td>
<td><code>public:</code></td>
<td><code>public:</code></td>
</tr>
<tr>
<td><code>A();</code></td>
<td><code>B();</code></td>
<td><code>A();</code></td>
</tr>
<tr>
<td><code>virtual void add();</code></td>
<td><code>virtual void add();</code></td>
<td><code>virtual void add();</code></td>
</tr>
<tr>
<td><code>void addAll(int all);</code></td>
<td><code>virtual int cardinality();</code></td>
<td><code>virtual int cardinality();</code></td>
</tr>
<tr>
<td><code>virtual int cardinality();</code></td>
<td><code>virtual int cardinality();</code></td>
<td><code>virtual int cardinality();</code></td>
</tr>
<tr>
<td><code>private:</code></td>
<td><code>private:</code></td>
<td><code>private:</code></td>
</tr>
<tr>
<td><code>int a;</code></td>
<td><code>int b;</code></td>
<td><code>int a;</code></td>
</tr>
<tr>
<td><code>A::A(){</code></td>
<td><code>B::B(){</code></td>
<td><code>A::A(){</code></td>
</tr>
<tr>
<td><code>a=0;</code></td>
<td><code>b=0;</code></td>
<td><code>a=0;</code></td>
</tr>
<tr>
<td><code>void A::add(){</code></td>
<td><code>void B::add(){</code></td>
<td><code>void A::add(){</code></td>
</tr>
<tr>
<td><code>a++;</code></td>
<td><code>b++;</code></td>
<td><code>a++;</code></td>
</tr>
<tr>
<td><code>void A::addAll(int all){</code></td>
<td><code>void A::addAll(int all){</code></td>
<td><code>void A::addAll(int all){</code></td>
</tr>
<tr>
<td><code>for(int i=1; i==all; i++){</code></td>
<td><code>for(int i=1; i==all; i++){</code></td>
<td><code>for(int i=1; i==all; i++){</code></td>
</tr>
<tr>
<td><code>add();</code></td>
<td><code>a++;</code></td>
<td><code>a++;</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>int A::cardinality(){</code></td>
<td><code>int B::cardinality(){</code></td>
<td><code>int A::cardinality(){</code></td>
</tr>
<tr>
<td><code>return a;</code></td>
<td><code>return b;</code></td>
<td><code>return a;</code></td>
</tr>
</tbody>
</table>

This example has been taken from [105].
the modified code have changed and therefore the original code has, de facto, been changed and should be just as thoroughly tested as the modifications.

**Example 4.4.2 (Easily Broken Code)**

The code construct below evaluates the result of a search on `customerNumber` to see if the number was found or not. The problem is that if code, say a counter like `cntNotFound = cntNotFound + 1` needs to be added to the `ELSE` correctly then some knowledge and three changes are needed. First the programmer must know that the `IF` has an `ELSE` associated with it. Not all `IFs` are associated with an `ELSE`, and when they are the `ELSE` can be far away, on another screen or on a different printed page from the `IF`. Second, the `ELSE` must be changed to a `ELSE DO` to open the control block. Third, the `ELSE` `DO` must have an `END` introduced at the close of the control block. Finally, the `cntFound = cntFound + 1` code must be added between the `ELSE` `DO` and the `END`. If any of these changes are not made the program will have errors.

**Code Implementation in: NetRexx**

```rexx
call search( customerNumber )
customerFound = false
if rc = 0
  then do
    customerFound = true
    /* When customers are found the customer processing starts */
    /* more code would go here */
  end
else
  call error( customerNumber )
```

### 4.4.1.8 Error Prone Coding

The way in which a program is coded can lead to serious problems during implementation. Some code constructs are prone to introducing new errors when they are changed, as shown in Example 4.4.2, or to misleading the programmer when they are
read, as in Example 4.4.3.\textsuperscript{34}

\begin{example}
\textbf{Example 4.4.3 (Difficult to Read Code)}

The two code examples below show constructs that will mislead a programmer when making a change.\textsuperscript{a} In the first case the indentation misleads the reader into thinking the all four statements are executed only when \(a=b\). In fact only \(c=d\) is executed in this case, the others are always executed. In the second case, the programmer is again mislead into thinking that all four assignments are executed only when \(a=b\). Here only \(c=d\) and \(e=1\) are executed when \(a=b\), the others are assignments are always done.

\textbf{Code Implementation in: NetRexx}

\begin{verbatim}
IF a=b THEN
  c=d
  e=1
  f=2
  g=3
\end{verbatim}

\textbf{Code Implementation in: NetRexx}

\begin{verbatim}
IF a=b THEN DO
  c=d
  e=1
END
f=2
  g=3
\end{verbatim}

\textsuperscript{a}These code examples come from actual industry systems, although they have been changed to show the problems more clearly and for anonymity.
\end{example}

\textsuperscript{34}The interested reader is referred to [100] for more information on this subject.
4.4.2 Observations

In order for code that is part of a dependency to be changed, all code participating in the dependency must be considered. The previous section, Section 4.3, identified dependencies as being difficult to detect unless a systematic comprehension strategy is employed. In fact, if the dependent code is outside the scope of where the systematic strategy is used the dependency is likely to go unnoticed and a change to the related code could introduce a new error. To make the implementation of changes less error prone dependencies should be kept local to parts of code where they can be detected using a systematic comprehension approach.

Beyond dependence, the code itself can cause problems. Some coding constructs do not take the form a programmer expects or are written in a way that is prone to introduce new errors.

4.5 Testing

When specifications are ambiguous maintainers must often make assumptions about what program behaviour is required. It is likely that the assumptions the maintainer makes will be different from those made by the initiator of the original change request. As a result the maintainer may be under a misconception about the requirements for a change and will introduce a new design error. Design errors occur when an implementation differs from its specification. Because design errors are made under a misconception they are extremely difficult to detect by the individual making the error - they are convinced they have made no error. But even when specifications are not ambiguous programmers may still make errors when implementing specifications. How can these errors be detected? They are usually detected through a collection of activities called testing.

Hetzol in [65] describes a test activity as being one that increases the assurance or
belief that a program performs “as it should”. He goes on to observe that some way of describing a program’s desired behaviour is implied by this description and calls this description a specification.

Testing can only seldom prove program correctness. Showing that a program matches its specification is to prove that the program is correct. Testing cannot always prove program correctness because the state space for most programs is much too large for this to be practical[139]35, inspiring Dijkstra to observe “program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence” [37]. The state space of a program contains all possible states the program can be in. A program state is determined by its data state. A data state is a member of the program’s data space. The data space of a program is all possible data values for all variables in the program[108].

Testing can estimate reliability. Software reliability is a statistical measure relating a system to the pattern of demands placed on it[125]. If software reliability is the likelihood that the software will not fail then software reliability is really the likelihood “of not encountering a sequence of inputs that leads to a failure” [139]. Test data can only try to anticipate what the input sequences that will lead to a failure might be and include those, however, test data cannot anticipate all patterns of input data.

Testing therfore, for this thesis, is defined as a set of activities, including debugging, intended to improve the trustworthiness of computer programs. As the complete state space of a program cannot be tested the best maintainers can do is to test until they are satisfied the program will behave as expected most of the time. The trustworthiness of software is the probability that no serious design errors remain after a series of randomly chosen tests [139]. But trustworthiness is also the product of the

35In practical terms this means a program with a single variable which has a type able to take integer numbers contained in two bytes of storage (int in ANSI C or FIXED BIN(15) in PL/1) can take on $2^{16}$ possible data values and a program with two of these variables could contain $2^{16} \times 2$ data values.
4.5 Testing

software being used and behaving as expected, under both controlled test conditions as well as in production. The more software behaves as expected the more trust in that product grows.

Historically, testing has been considered synonymous with debugging while, in fact, they are two distinct processes. Debugging starts with known errors and attempts their correction. Testing and debugging are related since testing uncovers inputs to the debugging process as an unhappy byproduct[65].

In the data processing industry testing activities take place in a sequence of stages. In each stage larger parts of the software are tested in an effort to show that program modifications have had the desired effect and that the resulting product is trustworthy. A typical sequence of test stages used in industry are the following:

1. Unit testing: *unit testing* is done on relatively small pieces of code, called a *test unit*, that can be tested independently of the rest of the system.

2. Integration testing: *integration testing* re-introduces the changed unit to the subset of the system from where it is invoked. This subset of the system is then tested with the changed unit.

3. Regression testing: During *regression testing* the whole system is tested with the changed unit to ensure that the new behaviour can be observed and that behaviour that should be unaffected by the change is as it was before the unit was introduced.

4. Function Testing: Testing that is carried out before acceptance testing to ensure the software performs with the added functionality as expected.

5. Acceptance testing: During *acceptance testing* the initiator of the change request checks to see if the program modifications meet expectations.
After each test, the results must be verified. A successful test is one in which, for a given set of input values, the desired output values are produced\textsuperscript{36}. If the output shows a discrepancy with the expected values than the program will fail the test verification and the programmer will typically return to the program comprehension stage of the software maintenance process to determine “where it all went wrong” (shown in Figure 4.1).

The following section will describe typical difficulties encountered during the testing of changes made during software maintenance.

\begin{example}{4.5.1 (Data States)}
A byte in Java\textsuperscript{TM} is an 8 bit integer that can take on 256 states. Although there are only two paths through the program there are many paths through the program’s data states, from \( p=1, b=false \) to \( p=2, b=false \) or \( p=99, b=false \) to \( p=100, b=true \). Each of these paths, through a sequence of data states is an execution. All executions of this program will have blackBox\textsubscript{m} return false, except 99. Implemented in: Java\textsuperscript{TM}

\begin{verbatim}
public boolean blackBox( byte p ) {
    boolean b = false;
    p++;
    if { p == 100 } {
        b = true;
    }
    return b;
}
\end{verbatim}
\end{example}

4.5.1 Difficulties

Testing of analogue devices can be based on interpolation where it can be assumed that if a device functions according to its specification at two close points it will also perform correctly at all points between\textsuperscript{139}. Unfortunately, software is not like that,

\textsuperscript{36}For this thesis success will be the absence of errors in a test case as opposed to their presence.
as is seen in Example 4.5.1, where all values of \( p \) will produce the same returned value except when \( p = 99_n \). The mathematical functions implemented in software are not continuous\cite{139} making interpolation invalid for software testing.

Since it is impractical to test all possible executions of a program perhaps maintainers can hope for better results by maximising the number of executions that are tested. But this assumes that all executions of a program are the same. In practice this is also not true, as is demonstrated in Example 4.5.1. In practice much time is spent “on the judicious selection of test cases”\cite{143} that will model the sequence of inputs to the software when in its production environment.

A test case is a unique combination of input data that will produce a possible execution. The following sections describe difficulties typically encountered during the selection of test cases and during verification of test results.

4.5.1.1 Lack of Specification

Test cases, once selected, are supplied to the software being tested as input data. The output produced is then compared to expected values. This process, however, relies on the assumption that acceptable values produced by the program can be differentiated from unacceptable values. This, the oracle assumption, is fundamental to software testing\cite{143}.

Since the specification states what the program is allowed to do without a specification there is no clear statement of what is acceptable behaviour for the program. Without a clear description of the program’s correct behaviour, “correct” behaviour is a speculation on the tester’s part, a violation of the oracle assumption.

4.5.1.2 Diverse Input Data

When the number of possible executions for a program increases generating test cases that test a program to the point where it can be considered trustworthy is
a monumental task. Frequently the sets of input data are large with dependencies between test cases making testing specific behaviours in the program very difficult. When the sets of output data are also large verifying the output against expected results can also become very difficult.

This increases the effort required to generate test cases to find undetected errors. Since there is a limit to the effort that can be spent on testing and since the number of test cases that must be tested to engender confidence in a piece of software is usually very large\cite{139} these units are typically not tested thoroughly. Mills observes in [28] “testing carried out by selected test cases, no matter how carefully and well planned, can provide nothing more but anecdotes”. No matter how thorough the testing, only a small fraction of all possible program executions will ever be tried.

4.5.1.3 Large Test Units

Testing is more difficult when those units under test are not easily isolated from the rest of the software. During unit testing this increases the size of the test unit and with it the number of executions needed to convince testers of the program’s trustworthiness. Larger test units are also likely to have more dependencies with other parts of the software. These must then be tested during test stages after the unit test.

4.5.1.4 External Influence

Very often software relies on inputs from outside sources that are not users. An example of this are dates and times used in periodic processes like a month or year end. When the date is relied on for processing and is secured directly from an external source, like a system date is, this can lead to difficulties for testers. As the system date cannot be changed by the tester the behaviour that has a functional dependency
to system date is very difficult to test\textsuperscript{37}.

4.5.1.5 Avoiding Undesired Event Detection

The detection and handling of UEs increases code volume, and with it complexity, which has the knock-on effect of increasing development time for programs. Estimates of code written for error processing and housekeeping\textsuperscript{38} range as high as 90\% \textsuperscript{100}. For this reason error handling is often neglected to reduce the effort spent during programming.

One such example is the use of runtime exceptions in Java\textsuperscript{TM} where UEs can be handled using exception handling programming constructs. An exception, or exceptional event, is defined as “an event that occurs during the execution of a program that disrupts the normal flow of instructions”\textsuperscript{18} and are catered for with a special type of object and language construct.

The special Java\textsuperscript{TM} object is the throwable object, an object defined as a sub-class of the \textit{Throwable} class\textsuperscript{39}. The \textit{Throwable} class has subclasses \textit{Error} and \textit{Exception}.

Errors detected in the JVM are UEs like no class definition being found when an attempt is made to instantiate an object. These errors typically have little to do with the problem domain and, therefore, are not usually considered part of the application code.

All \textit{exceptions} are usually considered part of the application code and are further subdivided into \textit{RunTimeExceptions} and non runtime exceptions. \textit{RunTimeExceptions} are UEs like null pointers. They differ from non-runtime exceptions in that runtime exceptions do not need to be explicitly catered for in the code\textsuperscript{40}. Non-runtime ex-

\textsuperscript{37}Thanks to \cite{61} for identifying this problem.  
\textsuperscript{38}This consists of tasks like closing files or freeing memory.  
\textsuperscript{39}Failing to do this will result in a compiler error when the object is instanciate with the throw statement explained latter on in Section 5.4.1.  
\textsuperscript{40}How exceptions are explicitly handled is described in Section 5.4.1.
ceptions, conversely, are reported up the invocation stack\textsuperscript{41} and must be explicitly handled at some point. If error handling code is not implemented for a non-runtime exception a compiler error will ensue. This makes it tempting to make all exceptions runtime exceptions. This is done by making all exception classes sub-classes of \texttt{RuntimeException}, thereby avoiding the compiler messages complaining of exceptions not being handled.

This tactic, however, only postpones problems arising from exceptions not being handled until the programs execution. Errors detected during the programs execution are reported up the invocation stack. If they are not handled explicitly by the application they must be handled by the runtime environment. Removing compile\textsuperscript{42} time errors in this manner postpones the problem to runtime and increases the burden on those who must test it.

4.5.2 Observations

Two points are clear from the above descriptions of difficulties during testing. The first is that testing is only as effective as the supporting documentation allows it to be because without documentation differentiating between acceptable and unacceptable test results are based only on assumptions made by the tester. The second is that testing without understanding the internal workings of a program, treating the program as a black box, is hopelessly inadequate due to the size of the data space. The internal workings of the program must be analysed and the knowledge used to reduce the necessary number of test cases, the size of each test case and the critical data in each test case. If this is done then it can be argued that if the program works for the test cases it will work in all other cases - the vast majority. The reasoning is simple, by thoroughly testing the parts of the program expected to be used in production,

\footnote{This term is defined in the glossary.}

\footnote{This term is defined in the glossary.}
the likelihood of an error in production is minimised. The intention is to use the time
the program spends in testing as effectively as possible.

If not understanding the internal workings of a program produces only poor levels
of trustworthiness while understanding the internal workings produces better results
then, “what forms of program structure, what elements of programming style, and
what forms of discipline, can we find for the benefit of the confidence level of our final
product?” (Dijkstra in [37]). This question is answered in the following chapters.

4.6 Update Documentation

When production software is changed the documentation that supports the software
must be kept consistent with the software.

4.6.1 Difficulties

The difficulties associated with updating documentation are those found in the doc-
ument review, with the following additions.

4.6.1.1 Poor Documentation Medium

The physical form the documentation is stored in has a direct effect on its maintain-
ability. A critical weakness for much documentation is that it is in a medium that is
expensive to modify. If the medium hinders the ability to keep documentation current
it will inevitably become outdated and lose its value to maintainers.

4.6.1.2 Difficult Documentation Distribution

The documentation medium must lend itself to easy distribution. Even if the doc-
umentation can be updated readily it is vital that the updated copy replace older
versions; otherwise these will contain dangerous misinformation.
4.6.2 Observations

It is fruitless to try to compel maintainers to update documentation if they see no benefit in doing so, particularly when the process of updating and distribution is inefficient. Updating documentation must be quick and easy with an effective distribution mechanism so that the cost of updating documentation is minimal while the benefit of doing so is high.

4.7 Conclusion

The difficulties identified in this chapter are symptoms of deeper problems. These deeper problems have also been explored.

All the sections clearly identify deficiencies in documentation as a major contributor to making software maintenance work more difficult. But Section 4.3, Section 4.4 and Section 4.5 imply that other characteristics in software may make maintenance more difficult when ignored. Section 4.3 identified delocalised behaviour as particularly prone to providing an erroneous understanding of software behaviour. Section 4.4 showed dependencies as causing problems during code modification while Section 4.5 identifies relationships in the data and the software as contributing to the effort needed for testing.

If the points identified in this section contribute to difficulties in maintenance then their common theme is in the area of connections. Connections, like those described in Section 2.2.1.2, exist where one part of the software relies on some other part of the software. Connections are critical where it is assumed that the aspect of the software relied on will not change. If connections cause difficulties in the software then it follows that controlling these connections will improve maintainability; what more effective means of control than rules? To this end, a clear set of rules for controlling connections in the software will be presented in the next chapter while a clear set
of documentation that will obviate the connections will be described in the chapter following.
Chapter 5

Design Guidelines

Two important sets of decisions that must be made during the software design process are: how to decompose software into parts and how these parts should relate to each other. These two sets of decisions are the primary means for controlling the connections in software.

Given the analysis in the previous chapter it is tempting to develop rules to cure each of the identified symptoms. Treating the symptoms does not guarantee a cure. To develop a cure requires basic rules that can be applied generally, it requires principles and practical guidelines with which to implement those principles.

Clearly, the goal of any set of principles that support software maintainability must be to make the job of maintaining software easier. What makes the maintenance job easier? The observations made at the end of each step in the software maintenance process presented in the previous chapter can be used to compile the following list of characteristics that make software maintenance easier:

1. Before the modification

   (a) The location of the change can be narrowed down quickly and without overtaxing cognitive abilities.

Jens Andexer

September 26, 2001
(b) The site needing change can be easily identified once the location has been narrowed down.

(c) The side effects of a modification are local to the part of the system being modified and easy to detect.

(d) The non-local side effects that do exist are easy to detect and are easy to track down.

(e) The system is physically organised so that parts of the software are located where they are expected to be.

2. Making the modification

   (a) It is easy to understand the coding of the existing implementation.

   (b) The number of additional changes needed as a side effect of a modification is small - minimum ripple effect.

   (c) The existing implementation has been coded so that it is resistant to introducing new errors.

3. After the modification

   (a) Identifying new test cases to verify the modification is easy.

   (b) New test cases to verify the modification are limited to a small number of variables.

   (c) It is easy to differentiate between acceptable and unacceptable test results.

   (d) Only a small part of the system needs to be retested.

In this chapter design principles and guidelines are introduced both to help make decisions about how a system should be divided into parts and to help make decisions about how those parts should relate to each other in a way that controls the connections in the software in order to make the maintainer’s job easier.

Jens Andexer

September 26, 2001
5.1 What to Make a Secret

If a system is to be broken into parts then what are the parts it should be broken into? Parnas, in [128], refers to one type of software part as *modules* and defines a module as “a work assignment for a programmer or group of programmers”. Applying this definition to the Object Oriented Paradigm means a module consists of one or more classes.

If the previous chapter is correct and connections in the software cause most maintenance difficulties then, by themselves, modules may not help maintainability. An arbitrary set of classes might make maintenance easier but without criteria for what classes are put into the set improvement would only be by luck and not by design. In order to design for maintainability some characteristics that make modules more maintainable must be established.

Since modules are a grouping of classes and since dependencies cause problems it follows that modules should group classes that are closely related and thereby keep the dependencies within the module. One characteristic of modules called *cohesion* is the degree to which tasks performed by a single module are functionally related [72].

What is a function? In the mathematical expression \( Y = F(X) \), \( F \) represents a function that transforms values of \( X \) into values of \( Y \). Therefore, \( F \) represents the transformation of input data to output data. A module also transforms input data to output data and a module also has a function. Cohesion, then, is the degree to which the parts of a module are related to its function, the degree to which a module is internally linked. Sommerville, in [162] describes cohesion as follows:

A module is said to exhibit a high degree of cohesion if the elements in that group exhibit a high degree of functional relatedness. This means that each element in the program unit should be essential for that unit to achieve its purpose.
Although cohesion is a desirable characteristic for a module to have it does not suffice as the sole criterion for decomposition. The specification of a module’s purpose is critical in deciding what program elements are “essential”. What is needed are criteria for defining a module’s purpose so that the resulting modules will be maintainable. Indeed what is needed “is a criteria to be used for decomposing systems into modules” [123] so that the resulting modules will be maintainable.

This section provides criteria for system decomposition using the method known as “information hiding”.

5.1.1 Information Hiding as a Decomposition Criteria

In general, encapsulation is the process of enclosing in a sealed unit. If a membrane surrounding a capsule were opaque then encapsulation could be considered the act of hiding. In software encapsulation is also the process of enclosing. What is being hidden is some information that should be kept secret. Therefore, in software, encapsulation is the act of keeping secrets or of hiding information. What information should be hidden?

The information that should be made secret are design decisions about the likelihood that some aspect of the system will change. The software designer must carefully consider likely future changes when identifying modules to prevent the delocalising effects of future modifications.

This section provides three general categories of design decisions that should be kept secret and guidelines for what makes a good secret in each category.

5.1.1.1 Hardware Hiding Modules

Hardware hiding modules are primarily affected by adaptive maintenance. When hardware is replaced or modified these modules may require change since their primary secrets are hardware and software interfaces.
The following are guidelines for what modules should be hardware hiding modules and what the secrets are that should be encapsulated.

5.1.1.1.1 Encapsulate APIs Most software systems must interact with other software including Operating Systems, Database Management Systems or Dialog Management Systems\(^1\). In Object Oriented programs interfaces to other software take the form of an Application Program Interface or API. The API is an encapsulation of some other software, software outside of the application.

The user interface is the part of the software system the user interacts with directly. In a networked system there can be more than one type of user interface, for example a Graphical User Interface\(^2\) (GUI) or a 3270 Terminal Emulation. To isolate the software system from changes in the user interface technology this technology must be encapsulated. This encapsulation would not be necessary if APIs did not change but this is outside the maintainers control. Consequently, encapsulating APIs is safer than their native use.

In the case study the Application Window Toolkit (AWT) was used to develop the user interface for the application. There is no guarantee that all users will always have access to a computer that supports the AWT so this has been encapsulated. (For details please refer to Section 5.1.1.2.1 for the next step in this encapsulation).

5.1.1.1.2 Make a secret of the Structure of Stored Data How persistent data storage is implemented must remain a secret. Since the value of a data element must, in some cases, persist for periods of time much longer than the time it takes for a program to execute, years in the case of financial records, a medium outside the program must be used. The storage service can be a buffer, file, a data base or some

---

\(^1\)This term is defined in the glossary.
\(^2\)This term is defined in the glossary.
other facility for data retention.

The previous section provides a guideline for encapsulating the interface to the storage service. But as important as encapsulating the interface to the service is making a secret of the database’s structure.

For example, a database can be encapsulated and the secret of this encapsulation might be the structure of the underlying database. Because of the previous guideline the implementation of the Data Base Management System (DBMS) is a secret. This is not enough. The underlying model of the database must also be made a secret. The software using this module must not know if the DBMS’s recall language implies a relational model, like that in \textit{DB2}, a hierarchic model, like that in IMS, or if the database is, in fact, a sequential file.

5.1.1.2 Behaviour-Hiding Modules

Behaviour hiding modules are likely to need change during adaptive maintenance and make secrets of changes likely to occur in the problem domain. These modules implement the Systems Requirements\textsuperscript{3} sections that describe the software’s required behaviour and, by doing so, determine the values to be sent to the virtual output devices provided by the Hardware-Hiding modules.

5.1.1.2.1 Encapsulate the User Dialogue

The User Dialogue is the part of software that describes the logic of the interaction between the software and the user. This software provides behaviour that implements, for example, what fields appear on a panel or in what order the panels are displayed and relies on the services of the hardware hiding module that encapsulates the API.

As this part of the system interacts directly with people in the problem domain it is a locus for change. Experience has shown that most changes to the software have

\textsuperscript{3}For details please refer to Section 6.2.2.1.
some manifestation in the user dialogue in custom built applications.

In the case study a user dialogue encapsulates a group of similar user interfaces. As there is more than one type of GUI there can be more than one encapsulation of GUI APIs\(^4\).

Notice, however, that the user dialogue is not independent of the User interface. The restrictions imposed by the user interface are such that the dialogue cannot help but be effected. For example, the AWT allows access to various fonts, colours and print sizes while a 3270 screen allows only 80 columns and 24 rows on which characters can be displayed. Clearly the GUI allows much more flexibility in the presentation of data and it would be impractical to impose the restrictions of a 3270 screen on a GUI. Therefore, a dialogue customised to the GUI environment has been written. Another different dialogue could be written to encapsulate the 3270 screen. (\textit{For details please refer to Section 5.2.1 .})

5.1.1.2.2 Encapsulate Domain Characteristics Outside the Owners Influence

Some characteristics of the problem domain cannot be influenced by the software owner. These characteristics of the problem domain must be encapsulated.

Taxation is a characteristic of most problem domains in the financial sector. For this reason how taxation rules are implemented should be made a secret.

5.1.1.3 Software Decision-Hiding Modules

Software Decision-Hiding modules will change when there are changes to design decisions based upon mathematical theorems, physical facts or algorithmic efficiencies. For example, if a file must be searched from for a pattern the search algorithm used would be this type of design decision.

\(^4\)For example the Java\(^T^M\) AWT or a GUI built for a particular purpose that has no standard API.

Jens Andexer

September 26, 2001
5.1.1.3.1 Variables are Always Private  Most classes require variables to retain their data state. The variables used by a class should always be private to the class. If other classes need access to the data in the variables this should only be done through access methods and never through the access rules.

5.1.1.3.2 Define Modules that are Complex Types  A data element is an association between a name reference and a set of storage locations. This association is described by a data definition or data declaration where the name reference is associated with a type. The data type is a description of the representation of the data and with it a description of how the data should behave - rules that dictate allowed changes in the data's value (For example, an integer cannot take on the value 1.5).

A data structure is a complex form of data, like a stack, that has sophisticated rules for how it can be behave. A data structure, its internal linkings, accessing procedures and modifying procedures should all be part of a single module [123].

Most applications need data that has more complicated behaviour than that provided by types native to a programming language. The complex data types needed by an application should be implemented only once and then used by the rest of the system. The implementation of each complex type should include all functions that can be performed on that type.

In financial applications, for example, types that represent monitory values in a variety of currencies. How the currency values are stored is the secret kept within the currency type. The operations that are allowed on currencies could then provide the correct validation checking, rounding and conversion calculation built into the type.
5.1.2 Implications for Maintenance

Information hiding reduces conceptual inconsistencies, like those described in Section 4.3.2.9, that result from ambiguities in a module's purpose. The three categories of modules unambiguously define the type of secrets kept by modules in that category. By knowing the type of secret a module keeps its purpose becomes clear. As a result the location of the change can be narrowed down quickly and without overtaxing cognitive abilities.

The purpose of keeping secrets is to keep changes that affect the secret local to the module. This, in turn, reduces the side effects of a modification and keeps them within the module. Unfortunately, modules must interact with each other and encapsulation is no guarantee that a secret will be kept.

5.2 How to Tell a Story but Keep a Secret

Once a system has been broken into parts it becomes necessary to allow the parts to interact. The protocol for this interaction is the interface\(^5\). Since it has been established that controlling interactions is key for maintainability controlling the interface is a key to maintainability.

One characteristic of the interface that serves as a measure of the interdependence\(^6\) among modules in a computer system is called coupling\(^7\). Myers in [113] describes coupling as follows:

The subject of module coupling is entirely concerned with inter-module relationships. Minimizing module coupling is a process of both eliminating unnecessary relationships among modules and minimizing the tightness of those relationships that are necessary.

\(^5\)This is described in Section 4.4.1.2 on page 80.
Although desirable, loose coupling is not enough. Even when modules are loosely coupled, changes in a module affecting the interface may still be delocalised to other modules through the interface modification. What is needed is a way of making the interface itself less prone to change, even when the internals of the module change.

Abstraction is the process of reasoning from detailed facts to general principles. A central feature of this reasoning is the removal of detail to produce a generalisation. Since generalisations are less likely to need changes, abstraction can be used to create interfaces that are even less likely to need changes. Where does one start?

The basis for software design consists of hardware interfaces, interfaces to other software and intermediate levels of the software design. The behaviour displayed by one of these will be referred to as the base machine. It is the set of behaviour on which an abstraction will be based. The abstraction from this base machine then results in a new programmable machine. The new programmable machine will be referred to as the virtual machine\(^7\)\cite{138}.

Remembering that abstraction is a process of removing detail, it follows that the state space for the virtual machine will differ from that of the base machine. This difference in the state spaces is referred to in \cite{138} as a loss of transparency. The loss of transparency means that there are states and state sequences that are possible by programming the base machine that are not possible by programming the virtual machine.

Consider Figure 5.1 where diagram a shows a representation of the top view of a four wheeled vehicle. The front wheels are used to steer the vehicle by means of four strings attached in such a way that the wheel can still rotate. Pulling on a single

\(^6\)Notice that interdependencies among modules is not restricted to interfaces. Dependencies could be through interactions that do not take place through the module interface like those through global variables.

\(^7\)This is not the JVM.
Figure 5.1: Base Machine

string causes a wheel to pivot about the point where it is joined to the axle. The driver has four strings available to steer the vehicle. Notice, though, that directing the vehicle is not an easy affair. The wheel positions b and c can be achieved, but so can d and e, with equal ease. The result of this configuration is a vehicle that is difficult to learn to control because the steering mechanism provides behaviour not conducive to directing it. What is needed is a mechanism that allows the wheels to only take on those positions that are useful for guiding the vehicle.

Now consider Figure 5.2 where a steering wheel has been added. The strings are now attached to the steering wheel in a way that allows the vehicle to be guided by turning the wheel in the desired direction. This abstraction from the base machine

---

8In positions d and e.
Figure 5.2: Virtual Machine

results in a vehicle that is easier to use because it does not allow undesirable wheel positions.

It is important to notice that the abstraction imposes a loss of transparency by filtering out d and e in Figure 5.2. By no longer allowing wheel positions d and e the abstraction provides, or exposes, only the behaviour of the base machine relevant for the task of steering.

In software the same effect can be achieved by encapsulation and exposing functions that allow that encapsulation to be programmable in a way useful to the application.
5.2.1 Abstract in Small Steps

Within a virtual machine the base machine is abstracted and the classes used to do this may have a close relationship with one another. Due to the close relationship the classes share within the virtual machine the close coupling of classes allowed by inheritance can be used. To make the use of inheritance safe, even here, the abstraction to create a virtual machine should be small. By making small steps the instructions provided by machines built early in the development process can be used for more complex machines later on. This means methods should be made as simple as possible and should add only enough behaviour to provide a new useful function.

Consider the creation of types in the case study. At the bottom of the types module are hardware hiding classes that encapsulate types provided by the javaMath package. The FinancialValue\(e\) class uses the services of the javaMath package to implement its type. The Currency\(e\) then uses the behaviour of the FinancialValue\(e\) as the base machine and implements the Currency\(e\) type.

5.2.2 Use Package Access for Inheritance

Using inheritance within a package has been described as safe above and the detrimental effects of public inheritance where described in Section 4.3.2.8. To keep the inheritance used to implement an abstraction safe, all but the methods that describe the interface should have only packaged, protected or private access. This keeps inheritance local to the package.

5.2.3 One Abstraction per Package

The package is a language construct used by many Object Oriented languages as a container for related classes. A single package should contain only those classes used in the process of abstraction described above. Therefore, the classes within a package
are only those needed to build an abstraction from one or more base machines to a new virtual machine.

5.2.4 Use Public Access to Define an Interface

Many Object Oriented languages force the use of public access to allow a method to be used outside of a package\(^9\). If the package is to be used as a container for a module the interface must be defined using public access rules. A package can include its methods in the module interface by making the method public. It was shown, however, in Chapter 4 that making methods public has negative implications for maintenance. These will be minimised in Section 5.2.5.

5.2.5 Use Private Access for Containment

A module's behaviour must be made available to that part of the system which requires its services. However, it was shown in Chapter 4 that deep inheritance can cause problems in maintenance. Therefore, the mechanism used to make a module interface available to the rest of the system must limit the interface's availability.

Unfortunately, there is no language construct that allows this type of scope limitation. In fact the Java Language forces unrestricted access to the interface because in order to make a method available outside a package\(^10\) it must be designated public. Once designated public, though, the method is automatically inherited by all descendants of the virtual machines interface. What is needed is a convention that limits how far inherited behaviour can be passed down through generations.

An alternative to inheritance is containment. The containment relation also allows access to a class’ methods but provides access rules for restricting the visibility of the

\(^9\)Or equivalent language construct.
\(^10\)This term is defined in the glossary.
used data and methods. By making contained classes private behaviour will not be passed up thorough the inheritance hierarchy and will stay local to the using class.

5.2.6 Nest Virtual Machines

When the base machines have been encapsulated an intermediate design has been developed. The functions available from the intermediate design can then be used in new abstractions to build new virtual machines. Once again the new virtual machine will contain a secret, but this time the secret is the programmable states of other virtual machines.

Section 5.1.1.1.1 recommends the encapsulation of the User Interface while Section 5.1.1.2.1 recommends the encapsulation of the User Dialogue. This is an example of an abstraction that takes several steps to complete. The behaviour of a GUI and of a 3270 Terminal Emulation screen are so different that the dialogue between users and the system through these media is affected. Therefore, a separate dialogue was written for the GUI and could be written for the 3270 screen. The important feature of these two dialogues is that they interact with the rest of the system through the same interface. The first abstraction uses the AWT\textsuperscript{11} as the base machine and abstracts to the Graphical Application Interface module. Then the Graphical Application Interface module is used as a base machine for the next abstraction to the behaviour provided by the Application Dialogue Interface. This behaviour is always the same regardless of the display medium.

5.2.7 Never Say Never

Each module should provide exactly the set of behaviour required of it by the rest of the system through its interface; no less but also no more. If additional behaviour is

\textsuperscript{11}This term is defined in the glossary.
needed it can be added with a minimal impact on the system by adding new methods to the interface. If methods in a module interface must be removed, however, the impact can be significant.

The Java interface is a named set of method definitions without implementations[18]. The problem with a Java interface is that when a new method is added all subclasses that inherit the interface become invalid. This makes it extremely difficult to add or remove methods from this language construct. The difficulty in finding all classes that inherit from an interface make it impractical for large systems[173].

5.2.8 Implications for Maintenance

Modules defined according to this and the previous section will have the following characteristics\(^\text{12}\):

1. Each module’s structure will be simple enough to be fully understood, thus reducing the programmers need for error prone comprehension strategies.

2. One module’s implementation can be changed without affecting other modules, reducing the ripple effect of changes through the system.

3. The most likely changes should be local to a single module, reducing the likelihood of changes delocalising through the system.

4. It should be possible to make extensive changes as independent modifications to different modules, reducing their risk.

5. Maintainers should be able to understand what a module does without understanding how it is done, thus screening them from the detail of the implementation.

\(^{12}\)Derived from [142].
6. Maintainers should be able to identify modules relevant to a change and disregard irrelevant modules, using the as-needed strategy of comprehension to their advantage.

7. Lower level modules can be used by a variety of higher level modules and, in fact, must function without the higher level being present. This allows the reuse of lower level modules and simplifies unit testing.

“This means a good module is simpler on the outside than on the inside, and that a good module is easier to use than to build.” [67]

Although all the above are desirable characteristics on their own they will only help the maintenance of modules, not necessarily of whole systems. As described in Chapter 4 the unrestricted use of modules is detrimental to maintainability. What is still missing are rules for how to relate modules to one another.

5.3 A Hierarchy of Secrets

Designing software that has desirable coupling and cohesion characteristics is not enough. As observed in Section 4.4 the relations between parts of the system can result in a structure that hinders maintenance activities. Many of the problems described in Section 4.4 have their origins in the relationships between modules and not in the module design.

This section provides guidelines that describe how modules should relate to one another.
5.3.1 Relate Programs in a Uses Hierarchy

The "uses" relation, as described earlier in this thesis, exists when a module requires the presence of another module in order to function correctly while a hierarchy$^{13}$ is a loop free relationship between modules. Remembering Section 5.2, the software design starts with a base machine. When the base machine is external to the software system being designed and its encapsulating module uses no other modules within the software being designed then this module can be assigned a level 0. Level 0 has modules that use no modules within the system being designed. The next level, level 1, then "uses" modules from level 0, but from no other level. A hierarchy is ensured when modules on level $i_v$ use only modules on level $i - 1_v$ or lower. By restricting the uses relation to a hierarchy problems arising from control dependencies like those described are less likely.

5.3.2 Align the Invocation Hierarchy with the Uses Hierarchy

Programming languages do not have a programming construct that specifically implements the uses relation, instead there are many programming constructs that allow one module to use another. The most commonly used programming construct that makes a uses relation possible is the invocation relation. This does not guarantee a uses relationship between the invoking and the invoked programs since the invoked program may not be relied on by the invoker to complete correctly. Therefore, wherever possible, when a method is invoked the invoking method should rely on the invoked method for its own correct execution.

Notice that there are some cases where the invocation relation cannot be aligned with the uses relation. One reason is that there are more ways to pass control to

---

$^{13}$This term is defined in the glossary.
5.3 A Hierarchy of Secrets

a method than by invocation. Class creator methods are invoked implicitly during a class’s instanciation. Therefore the new operation, used in most Object Oriented languages to instanciate an object, binds the creator method into the uses relation. In a similar fashion the throw operation in Java™ creates a uses relation between the method detecting an error and the creator method for the error object.

Another anomaly occurs when a method invokes another method but does not rely on the outcome of the method invoked. For example, software is often written with a debug mode that is not included as part of the program specification. When the debug mode is on, messages are written to a log or displayed on a screen. When the writing of these debug messages is done by a special purpose module then this module is invoked during debugging but is not relied on for the module to complete its task. The debugger is not part of the “uses” relation for the invokers because it is not part of their specification and it is not relied on to make the invoicing module correct.

5.3.3 Simplify Downward

Modules toward the bottom of the hierarchy should operate in simple data structures and be treated as if they were fixed. Modules closer to the top of the hierarchy can alter the data structures implemented in terms of lower level classes. The AuditedFinancialValue_e is an example of this from the case study. This is a complex data structure that adds an entry to the audit trail each time its current value is changed. Each entry in the audit trail is made up of a FinancialValue_e and a DateTimeStmp_e. The two lower level data structures are treated as if they are fixed by the AuditedFinancialValue_e although the next level allows for the creation of entities in the audit trail.

Jens Andexer

September 26, 2001
5.3.4 Use to Simplify

Although defining the uses hierarchy is useful what is needed is criteria for when one method can use another. The following criteria are tendered in [121] and have been adapted to the Object Oriented paradigm. The following conditions must all be true to allow A 'uses' B:

1. Method A is essentially simpler because it uses method B.

2. Method B is not substantially more complex because it is not allowed to use method A.

3. There is a useful subset containing method B that does not contain A.

4. There is no conceivably useful subset containing method A that does not contain B.

Notice that these criteria can result in methods that are part of the same class being in different levels in the uses hierarchy because methods in a class can use other methods in the same class.

5.3.5 Implications for Maintenance

H.A. Simon makes the following observation about complex systems in [154].

Intracomponent linkages are generally stronger than intercomponent linkages. This fact has the effect of separating the high frequency dynamics of the components - involving the internal structure of the components - from the low frequency dynamics - involving interaction among components.
Courtois in [26] makes two observations, based on Simon and Ando, about the dynamic process of computer systems\(^{14}\). First Courtois observes that "frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached.[27]." When the guidelines in this section are followed, modules will tend to aggregate according to Courtois observations. The lowest level will contain simple modules. At each higher level in the hierarchy modules benefit from using modules lower down and become simpler themselves.

Courtois goes on to observe that "in general, interactions inside subsystems are stronger and/or more frequent then interactions among systems [27]." This type of module structure reduces the impact of changes and when changes are needed their impact is more predictable. When a module is changed in a hierarchical structure only modules above it will be affected directly by the change. This reduces the likelihood of changes rippling through the system and reduces the number of modules that will need to be tested.

The ordering of modules in this type of hierarchy also provides a simple but useful heuristic for maintainers when narrowing down possible locations for a change; more complex modules are up in the "uses" hierarchy, simpler modules are down.

5.4 Designing in the Presence of Undesired Events

Software reliability has been referred to as a statistical measure relating a system to the pattern of demands placed on it[125]. A system is considered reliable when it is highly probable that, when a demand is placed on it, the system will perform

\(^{14}\)These observations were made by Simon and Ando in [155] about complex systems of data in general and called this type of system nearly completely decomposable. A system would be completely decomposable if there where no interactions between the subsystems.
satisfactorily. If satisfactory behaviour is taken to mean not ending with an unanticipated UE then reliability becomes the extent to which inputs are anticipated in the program code.

Programming to anticipate inputs comes at a cost. Code written for detection and handling increase program code volume and complexity\textsuperscript{15} and, therefore, UE detection and recovery must be considered in a design for maintainability.

UE recovery has two aspects. First, programs must be written to provide responses to undesired events while, second, people must often respond to UEs detected by the system as well. This section provides guidelines for both these sets of responses.

5.4.1 Providing for a Program’s Responses to Detected UEs

Once detected a program should provide a response to an undesired event. This section provides guidelines for the programs response to UEs.

5.4.2 Provide Sufficiency in UE detection

The UEs that a program detects should be sufficient to guarantee that if no UE is detected the machine can provide the requested service without limitations. This means modules must validate their parameters to ensure the data supplied to them is acceptable for the requested task.

Consider the FinancialValue\textsubscript{v} class. This class can accept character strings and ensures these are valid for translation into a decimal number. The validations are sufficient only to guarantee a successful translation from a character string to a decimal numeric, no more and no less.

\textsuperscript{15}For details please refer to Section 4.5.1.5.
5.4.2.1 Propagate UEs upward

UE propagation can go either downward or upward in the uses hierarchy. Propagation downward means a lower level in the hierarchy would have knowledge of the state of the higher level module, its error state, leading to a violation of the principles introduced in Section 5.1.1. Consequently UE propagation must only be upward in the uses hierarchy.

5.4.2.2 Separate UE Recovery from normal case code

Modules should be written so that code for UE detection and recovery is separate for the normal case code and can be modified independently. This is effectively done using a software analogy to a trap in hardware. Most computer hardware is designed to detect some common UEs like division by zero. In the event of such a UE, control is passed to a specified location for special handling.

Many Object Oriented programming languages have equivalent programming constructs to the trap. Java\textsuperscript{TM} provides for this with throwable classes\textsuperscript{16}, along with the following statements for UE reporting and handling:

- \textit{throw} The throw statement instanciated an object that must be an instance of a subclass of the throwable class\textsuperscript{17}.

- \textit{try} The try block identifies critical code in which an exception can be detected, thereby defining the scope of code that can handle the UE.

- \textit{catch} The catch block identifies the code to handle exceptions reported in the try block.

\textsuperscript{16}described in Section 4.5.1.5.
\textsuperscript{17}For details please refer to Section 4.5.1.5.
• *finally* The finally block is always executed regardless of what happened in the try block. This provides an opportunity to complete processing that is common to exceptions that are caught, to exceptions that are not caught\(^{18}\) and to non-exception completion.

Using these constructs allows the normal case code to be included in the try block and the exceptions to be handled in catch blocks.

### 5.4.2.3 Really only on information local to the detecting Module

When a UE is detected it is the result of an event that takes place in the detecting module. Modules the detecting module relies on are unlikely to have information that will help in the handling of the UE. Because the detecting module is an encapsulation it will have no information about its invoker available either. As a result UE response software should be written to depend only on information local to the module in which they are detected.

### 5.4.2.4 Write Error detection and Recovery to allow enhancement

The UE detection and recovery software is specific to a module and should reside with it. The objects thrown as a result of a UE, however, should be encapsulated in package of their own. In this way UE reporting is standardised, yet flexible and expandable because it uses standard error objects while the UE handling code is special purpose, developed specifically for the detecting module.

### 5.4.3 Providing for Human Responses to Detected UEs

When software cannot recover from a UE human intervention is likely necessary. This section provides guidelines on how some of the information needed by programmers

\(^{18}\)Not all exceptions that can be detected in the try block need to be handled in the catch block, some may be passed on up the invocation stack.
during corrective maintenance operations on unrecoverable UEs can be provided by the software.

5.4.3.1 Retain the trail of Propagation

Part of the information needed by programmers after an unrecoverable UE is what steps the software took during its recovery attempts. This information can be provided as the UE is reported up through the invocation sequence as successive efforts at recovery are made. It is, therefore, important to retain the sequence of propagation and a record of recovery efforts.

For example, part of standard error handling in Java™ should be retaining the sequence of propagated exceptions. When an UE is detected an exception is thrown which passes control up the invocation sequence to a point where the exception is caught. When the exception is caught special processing attempts to handle the exception. If the special processing fails to handle the exception a new exception is thrown, but the association between the caught and thrown exceptions must be retained.

5.4.3.2 Use UE Locators

In large software systems the same UE can be detected in more than one place. This can make locating where in the code text a specific exception event occurred a difficult task.

In Section 4.3 on Program Comprehension one of the key pieces of information needed during corrective maintenance was identified as the location in the code where the UE was detected. In the observations to the Program Comprehension section it was noted that finding key code fragments is a difficult process. Even in well documented systems associating a message produced when an UE is detected to its documentation and to the location where the UE was detected in the code can be
ineffective.

To help in this process the location where an UE is detected in the code text should be retained using an error locator. An error locator is a key that uniquely identifies the site in the code text where the UE is detected. Therefore, the error locator is a beacon that consists of a key that uniquely identifies each UE detected in the software. Additional information can be provided for example, the computer where the method was running and the session on the computer if this information is available.

UE locators, in conjunction with the trail of propagation, help to associate UE detection with its correct reporting and handling. By identifying the location in the code text where the application detected, reported and attempted to handle the UE in the trail of propagation the programmer can determine if the UE was reported and handled correctly.

5.5 Guidelines for Maintenance-in-the-Large

Making software that is maintainable and keeping software maintainable requires a good design and good documentation. On a daily basis it also requires doing a lot of little things right and doing so consistently. This section identifies some of the little things that should be done and provides guidelines for doing them right.

5.5.1 Programming Standards

Central to the process of program comprehension are the limitations of human cognition. To design programs for maintainability requires taking into account the cognitive limitations of programmers who will perform maintenance tasks on the programs.

Miller in [106] places limits on human abilities to process information. He also introduces a means of increasing this limit using a process called recording. Recording
is simply the process of mentally encapsulating detail and forming it into a chunk. For example, when programmers are learning new programming language syntax they may read the code as a series of characters where each character is a separate chunk. Almost immediately they identify groups of letters as key words and then the programmer can deal with key words as chunks. The keywords can be chunked into control structures and the control structures can be chunked into code fragments in a process that creates larger, more abstract chunks from smaller ones [106][169]. Identifying chunks for abstraction is made more effective when programmer expectations about the chunks are met. Meeting programmer expectations requires the consistent use of control structures and consistent program presentation so that chunks can be effectively identified.

Programming standards formalise rules of programming discourse\(^{19}\). To do so they must regulate how program constructs such as loop and conditional statements should be programmed. Included in *programming standards* are *coding standards* which describe the use of white space and comments. The goal for using these two types of standards is to allow programmers to effectively recognize programming constructs and, therefore, allow these to be abstracted into chunks more easily.

This section provides guidelines that should be considered when programming standards are developed.

### 5.5.2 Promote close Adherence to Programming Standards

It is worth while motivating programmers to adhere to programming standards. This goes beyond aesthetics, [161] and [104] showed that consistency in program layout is important to program comprehension. In [161] Soloway *et al.* go on to show that even small deviations from the norm have a significant negative effect on program

\(^{19}\)Introduced in Section 4.3.1.5.1.

Jens Andexer  
September 26, 2001
comprehension, particularly on experienced programmers. More important than what standards are used is that they are consistently applied.

5.5.3 Avoid Very Small and Very Large?

[20] shows that arbitrary limitations on sub-routine size\(^{20}\) can unnecessarily increase the development cost of a sub-routine\(^{21}\).

De-emphasising method size does not mean that smaller is always better. In [62] Hatton found that very small and very large methods had higher error densities than medium sized methods. However, the relationship between code size metrics for modules and defects appears, from a survey of the literature, to be poorly understood\(^{22}\). Indeed, size has not been included as a criteria for decomposing systems into modules and, therefore, should not be used as such. Module size is, therefore, left to the discretion of the software designer.

5.5.4 Promote Single Entry, Single Exit

In [37] Dijkstra advocates the convention of a single entry and single exit point to sub-routines and loop constructs. This means that a subroutine or a loop has only one point where control can enter the construct and one point where control can leave the construct. In [160] Soloway showed that exit points from within a loop can lead to improved comprehension in inexperienced programmers although experienced programmers prefer single entry and exit constructs.

The problem with single entry and single exit is that it can lead to complicated exit conditions. It is often easier to break an exit condition apart and provide exit

\(^{20}\)Calculated using lines of code (LOC) as the metric.

\(^{21}\)This study was done on FORTRAN programs but because a sub-routine is a, hopefully, logically related unit of code this author believes it is safe to extrapolate to methods in the Object Oriented paradigm.

\(^{22}\)[62] [54] [25] and [95] argue there is a relationship while [7] [151] and [46] argue their is none.
points for each of the constituent conditions. In Example 5.5.1 a convoluted set of conditions is used to differentiate between sums assured that need to be added together for a total sum assured. The two conditions in the SELECT block could be consolidated into the exit condition of the loop but this would make the loops exit condition difficult to understand. This is a poor candidate for single entry single exit.

Control blocks, like the SELECT in Example 5.5.1, can and often do go over several pages. When this happens an exit point is easily missed during code analysis, very likely resulting in errors in the analysis.

The convention of single entry and single exit points should be a goal but not slavishly followed. How complicated the exit condition can become before it is unacceptable is a decision the programmer must make. When exit points from within programming constructs are used then these must be clearly marked in the comments.

5.5.5 Use White Space Consistently

The non-text portion of the code consisting of spaces, tabs, blank lines and line breaks, is called white space. How white space is used to show relationships in the code is referred to as program layout and the objective of layout is to make program code easier to read. A good program layout will improve readability by associating logically related parts of the code in a consistent manner. A good layout will also help to maintain readability after the code has been modified and to help identify incorrectly ended control structures.

An important aspect of indentation is that code related through a common control structure be indented the same amount, optimally between 2 to 4 spaces[104]. In the case study each “{” is placed at the end of the opening statement of the control block and is matched with a “}” indented 3 spaces\textsuperscript{23}. The code related to the control

\textsuperscript{23}See the coding standards in the Standards Manual.
Example 5.5.1 (A Complex Set of Exit Points)

To move all the exit conditions into the WHILE would make its condition difficult to understand.

Code Implementation in: Net REXX

n=0
sumOfSumAssureds = 0
DO WHILE( policy.n.type != "" )
  SELECT

    WHEN policy.n.type = 1340
    THEN DO
      IF sumAssured( 1 ) > 10000
        THEN
          LEAVE /* exit WHILE loop */
    END

    WHEN policy.n.type = 1341
    THEN DO
      IF policy.n.sumAssured > 15000
        THEN DO
          IF policy.n.multipleDwellings = true
            & policy.n.sumAssured > sumAssured( 1 )
            THEN
              NOP
          ELSE
            LEAVE /* exit WHILE loop */
        END /* if */
    END /* when */

  OTHERWISE
    NOP

END /* select */

sumOfSumAssureds = sumOfSumAssureds + policy.n.sumAssured
n=n+1
END /* while */
structure is indented three spaces and new control structures opening inside the open control structure are indented a further three spaces.\footnote{For an extensive handling of Program Layout the interested reader is referred to chapter 18 of \cite{100}.}

5.5.6 Use Beacons

Most of a programmer’s experience resides in long term memory in the form of plans. The process of recalling this knowledge is expedited when it is triggered. One type of trigger is called a \textit{beacon} or text strings that can act as cues to index into long term memory\cite{169} triggering the recall process. These text strings provide hints about how externally visible behaviour is implemented in the code text\cite{148}.

Beacons are an effective way to provide a traceable connection from the users of the software to the software itself. When a beacon is visible to users and repeated in both the documentation and the code text it can provide a traceable association between the user’s view of the system, the implantation of the functionality and its documentation.

A traceable connection of this sort can be a unique screen identifier that is visible to the user or a unique UE identifier that is reported with an error message. This type of identifier provides a locus around which the user can supply additional information. Beacons can save a great deal of effort associating the software’s behaviour as observed by a user with a program’s documentation and code.

5.5.7 Keep Naming Conventions Simple

Names can serve as beacons to help trigger long term memory, thus helping in the association of meaning to the named entity. It is, therefore, important for a naming convention to support accurate naming but it is also important for the convention to be simple\cite{181}. The names selected for classes, methods, variables, etc. have a
subtle effect on the thinking of the reader[182]. This prejudice in thinking can lead to misinterpretation where programmers reading the code text associate a chunk with the wrong mental plan[24].

The convention applied in the case study used relatively long, descriptive names for classes and class or instance variables while variables local to a single method had simple names. The class and variable names are prefixed by a verb to produce the method names. In addition classes are capitalised while variables and methods are not\textsuperscript{25}.

In the case study constants have long descriptive names and are used instead of all in code literals, for example using “nine” for the constant “9”. This in effect uses the compiler as a spell checker for constants. If there is a typing error in a constant the compiler will identify it as a error. If the constant were a literal a typing error could go unnoticed.

\section{Organize Programs According to Uses Structure}

Lakhotia in [85] identifies providing information where the programmer expects to find it as one of the most important aspects of program comprehension. For this reason the physical organisation of the program source code into files should, as closely as possible, reflect one of the software’s structures. A common choice of structure for this purpose is the invocation relation because of its familiarity to programmers.

\section{Generalize instead of Clone}

There are no clear rules for when to clone a program and when to adapt an existing program to include new behaviour. Experience has shown that in a well structured system adapting an existing program is almost always more attractive than cloning

\textsuperscript{25}For an extensive handling of naming the interested reader is referred to chapter 9 of [100].
because in well structured systems the adaptation represents less effort. In poorly structured systems cloning is the more attractive alternative, particularly when coupled with the justification that the clone needs less testing.

5.5.10 Provide a Test Environment that is as Close to Production as Possible

As pointed out in [158] the key to success in corrective maintenance is recreating the problem. Ideally problem recreation is in a test environment which has been populated with the production data that accurately portrays the unacceptable behaviour. This requires a mechanism for identifying\(^{26}\) and copying related production data to the test environment.

5.5.11 Minimise the Depth of Inheritance

It has been recommended above that abstraction from a base machine to an abstract machine should be done in small steps. Both [29] and [41] argue that deep inheritance is confusing during maintenance. In the absence of other clear criteria for limiting the depth of inheritance the recommendation in [29] to limit inheritance to about 3 levels provides a clear and acceptable guideline.

5.5.12 Implications for Maintenance

Maintainability is not strictly quantifiable. Even the definition of maintainability (Definition 2.3) reveals an element of subjectivity. It seems that software maintenance is plagued with a human factor, the maintainer, and the maintainers limitations. Throughout the section on comprehension recurring themes were programmer limitations and programmer expectations. This section has provided guidelines that address human factors in software maintenance.

\(^{26}\)For details please refer to Section 5.4.3.
5.6 Conclusion

In the introduction to this chapter a list of characteristics that make software maintenance easier was presented. The sections in the chapter have presented principles and guidelines that help make design decisions for decomposing into parts and relating the parts of a software system so that these characteristics are present.

What is it about the presence of these characteristics that makes software easier to maintain? What are the underlying qualities in software that make software easier to maintain, indeed what qualities can be used as goals when designing for maintainability? The following is a list of these goals. The list has been compiled from a survey of the literature and from personal experience.

**Intellectual Manageability**

Intellectual manageability means that understanding enough of a software system to successfully make a modification does not overtax human cognitive abilities. Korson in [83] argues that limiting the amount of information a programmer must process during maintenance tasks yields significant benefits and goes on to empirically show positive effects of Information Hiding on maintainability.

**Flexibility**

Flexible software is easily changed, allowing it to be adapted to new situations with a minimum of effort. In [121] it is observed that a clever designer can design for flexibility without a “significant” run-time\(^{27}\) cost but at the expense of effort during the software design.

**Generality**

General software can be used in a variety of situations without needing changes. This attractive sounding characteristic has two benefits that derive from Fisher’s “Fundamental Theorem” which states “the better adapted a system is to a particular

\(^{27}\)This term is defined in the glossary.
5.6 Conclusion

environment the less adaptable it is to new environments.” The first benefit is that general software is less likely to need change and, therefore, suffer from its effects. The second benefit is that if the software does need modification then change is likely easier made. These attractive characteristics, however, come at a price. Parnas observed in [121] that there is an inevitable run-time cost to generality.

Reuse

A close relative of generality is software reuse, a characteristic where the same part can be used by other parts of the system and, therefore, only needs to be developed once. Reuse does not only save development effort. When software is used, “bugs” are removed as part of the corrective maintenance effort. Therefore, in the absence of enhancement the number of bugs declines during the software’s maintenance history and the software becomes more trustworthy as a result. The benefits of this are described by Gall in [55] as follows:

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never\(^28\) works and cannot be patched up to make it work. You have to start over, beginning with a simple working system.

The benefits of reuse are clear if the word “work” is interpreted as “trustworthy”. To develop large systems that are trustworthy requires parts that are trustworthy. Since trustworthiness\(^29\) is the product of the software functioning as expected, and doing so for some period of time, reuse is a necessary part of building complex systems cost effectively\(^30\).

Predictability

\(^28\)Any system can be made to work given enough effort is invested. The issue is how much effort must be invested to make a system work.

\(^29\)For details please refer to Section 4.5.

\(^30\)The “cost quality” benefits of reuse were found empirically in [20].

Jens Andexer  
September 26, 2001
Central to software maintenance is the maintainer. The more a software system is written so that a maintainer finds what is looked for where it is expected to be the easier the software is to maintain.

**Code Robustness**

Some code constructs are misleading during comprehension and are prone to new errors when they are changed. When code constructs are used that make what they implement easily recognisable and that are resistant to new errors the maintainer’s job becomes easier.

Structuring software for maintainability is only the beginning of making, and keeping, software maintainable. It has been shown that program code is a poor medium for communicating information. Because of the entropy in using program code as a communication medium, a programmer’s understanding of the dependencies in the software, and in the problem domain, is very likely to be incomplete. An incomplete understanding of the software’s dependencies makes the introduction of new errors very likely and also contributes to the software’s deterioration.

The answer is clearly to provide a medium for communicating detailed information about the software and about the problem domain that is more effective than program code. Documentation is such a medium.
Chapter 6

Documentation Paradigms

In [108] Mills refers to a rational program as “one whose internal mechanism is transparent to some set of people; a natural program is one whose internal mechanism is known to no one.” The terms Mills uses, *rational* and natural, are subjective and describe the ends of a spectrum of partially understood programs. To maintain a program, the part needing modification must be understood, along with those parts sharing dependencies with the modifications. Therefore, a tendency to the rational, as opposed to the natural, is a necessary prerequisite for maintainability.

The difficulties described in Chapter 4 are symptomatic of systems tending toward a natural state. As software systems tend towards a more natural state they become more difficult to understand and it requires more effort to make changes. Testing the changes at this point also becomes more time consuming. A worthy question is how can rational, maintainable, software be designed and, once built, how can the software be kept rational?

In [132] Parnas refers to a rational design process as one where a program can be derived systematically from a precise statement of requirements. He goes on to point out, however, that “We can never find a process that allows us to design software in a perfectly rational way”. The software development process is spared none of the
impetus for change present during maintenance. It is this dynamic that plays the primary role in preventing a rational design. How can a rational design, and with it rational software, be produced by an irrational design process? - By faking it.

Even if a rational design process is not possible the deliverables of the design process can be made to appear as though they were produced by a rational process. A rational design can become natural if it is not properly maintained, therefore, the maintenance process must also be rational. A rational maintenance process is one that contains rational redevelopment and redesign of the software product. If the faked rational design process is not continued its benefits will disappear and the software product will tend to become more natural.

Documentation plays a vital role in both the development and in the maintenance of computer systems. The value of documentation in the “design through documentation” [136] concept is clear although its value in the maintenance of a computer system is often underrated by maintainers.

This chapter will describe the role documentation plays in software maintenance and what can be done to make documentation more effective for maintainers. Then the sources of documentation will be presented and the value of documentation to the maintainer will be described.

### 6.1 The Role of Documentation

Software documentation fulfills two roles. First, documentation serves as a storage medium in which large amounts of information can be retained and from which the information can be retrieved. In its second role documentation is a communication medium for detailed information about a software product. This makes documentation the sole forum where a problem and its solution can be retained with clarity and

---

1 the benefits of doing this and the process itself are described in [132].

---

Jens Andexer  
September 26, 2001
6.1 The Role of Documentation

precision, giving documentation a central role in the development and maintenance of a software product.

Documentation, regardless of its physical form, is a medium well suited to delivering a detailed message to its audience. Documentation is most useful during maintenance when it is effective in its storage and communication roles. This section describes characteristics of documentation that make it effective.

6.1.1 Subsetable

In the previous chapter guidelines were introduced that make it possible to treat parts of a system as independent. Because these parts are independent they can be used from different parts of the same system as well as from different systems. Therefore, for systems to be truly modular their documentation must also be modular.

6.1.2 Precise

The information contained in software documentation will be used to design, implement or maintain the software product. If documentation is not precise the reader must make assumptions to fill in the missing details. If, for example, the requirements are imprecise about the data to be produced in a report then the designer of the report must make assumptions about the data to be outputted.

The assumptions which must be made when documentation is imprecise give latitude for errors to be introduced by those relying on the documentation. Errors introduced in this way are difficult to detect because the validation of a software design or implementation relies on detailed supporting documentation.
6.1.3 Complete

The documentation must be a complete description of the software. Omissions will again allow programmers to make assumptions giving the potential for errors.

6.1.4 Current

Keeping documentation current should be a byproduct of its use. Documentation tends to be used when it is the most effective source of information. Thus software documentation must be made up of working documents. A hallmark of effective working documents is that they are used and that those using the documents keep them current in order to keep them useful. Therefore, there is interaction between the users of the documents and the documentation.

McLuhan [101] calls this “warming the medium up”. This involves increasing the interaction between the user of the medium and the medium itself. By actively involving the document users in a formal review process\(^2\) as well as in a continuous informal review process the medium is warmed up. The reader becomes a more active participant of the documentation process and reader are motivated to participate in the process of keeping the documents useful when they rely on the documentation for their own effectiveness.

6.1.5 Verifiable

“Verification is the task of showing that a work product is free from errors ” [66]. In this chapter the work products are documents and documentation can most effectively be verified through reviews. A review is a detailed examination of a work product to identify errors. Reviews are typically used to verify the documentation produced as part of the development process but can be equally applied to documentation

\(^2\)The review process is not discussed in this paper but is discussed in detail in [141].
produced and modified during software maintenance.

Errors that are detected during review can be classified as follows:

1. **Omissions** - things left out of the document completely

2. **Infeasibilities** - behaviour that is impossible for the software to deliver.

3. **Inconsistencies** - places where an aspect of the system will not work \[141\].
   For example if a value like “customer cash account balance” has more than one meaning.

4. **Inefficiencies** - places where the design imposes inefficient behaviour \[141\].

5. **Ambiguities** - situations where the documentation can be interpreted in more than one way \[141\].

6. **Inflexibilities** - situations where the design of the software does not accommodate change \[141\].

### 6.1.6 Standardized

The documentation for a computer system cannot be seen as a set of single, disjointed reports. A system’s documentation must be integrated to form a cohesive set of manuals in which relevant information is highly accessible.

Standards in documentation within a single system and among systems within an organization will help towards consistency in the presentation of information and thereby its accessibility. This consistency will reduce the time spent becoming familiar with different documentation styles allowing the user to become expert in using the standard forms.
6.1.6.1 Content

The following section provides some of the key aspects of documentation that need to be considered when developing documentation standards.

In an integrated approach each document has a clear purpose. Standardized contents can be specified so that there is a minimum of overlap between documents and so that documents produced later in the development process build on documents produced earlier on. By standardizing the contents of each document readers with a specific need for information know exactly where to go first.

6.1.6.2 Language

Documentation for industrial applications requires a balance between formality and informality. Precision in documentation is best achieved by using the language of mathematics with formal methods. Industrial software professionals, however, find this type of documentation difficult to read and to understand, preferring to rely on less precise natural language prose. There must be a balance between formal and informal methods which makes the documentation precise but also accessible.

Once the uneasy balance of formality and informality has been found standards in language usage for both natural language and in mathematics must be set to ensure consistency. Software development projects quickly develop a culture and a language based on the application area. It is important that acronyms and words with special meaning in the application area be clearly and precisely defined. An example is the acronym IMS, which, in the data processing context describes a data base management system. In an application area the same acronym can be interpreted as Information Management Services, a department for whom software is developed. This makes a statement like “We need to let IMS supply that data” dangerously ambiguous.
Without a clear definition of these terms this ambiguity can lead to errors.

Since mathematics is needed for precise documentation it is important that the mathematical notation is standard not only within a single system but also among different systems. For example, in [64] a standard notation is introduced which is used for the requirements of the A7E aircraft. This notation is also used by van Shouwen in [167]. Users of both these documents need learn only one set of notation to use both documents, making the entire set of documents more accessible.

6.1.6.3 Format

Consistency in the presentation of information in documentation is a means for improving comprehension. When documentation is in a standard format the user only needs to spend time becoming familiar with how a documents' contents are presented once. This has significant benefits in industry where one individual is often involved with more than one software system or in several development phases for the same system.

6.1.6.4 Media

Technology brings with it a great variety of media in which documentation can be held, each of which has its own advantages and disadvantages. If working documents, like those described above, are used then the maintainability of the documentation becomes an issue. To keep documentation current it must be in a form which allows easy access for change and easy distribution of new versions.

In Example 6.1.1 a system of documentation is presented that is effective but relies on no specialized technology. In this case the medium of the documentation fits its needs well because the medium is simple; it can easily be changed and the new version can easily be distributed.

---

3Indeed, in the A7E documentation IMS means Inertial Measurement System[64].

---

Jens Andexer  
September 26, 2001
Example 6.1.1 (Documentation)

An organization has software specifications on paper ring binders and written in pencil. In order to change to the specification the correct ring binder must be removed from its shelf and logged out, the original specification photocopied and retained and the modification to the specification made by erasing the old and pencilling in the new specification. The new specification is then photocopied and given to the programmer to work from when modifying the code. After the specification change the ring binder is logged in and returned to its shelf.

This seems an arcane system but, in fact, it works very well. Because the original specifications are centrally located they are accessible to everyone who needs them. There is only one copy of the original so it is always the most up to date. The changes to the specifications are easily made and once done the specifications are again accessible to everyone who relies on them in the most current form. The programmer can work from the most current copy of the specification without restricting access to the entire document. Finally, this medium requires no special technology or expertise.

6.1.7 Many Audiences, One Set of Documents

As a medium for communication, software documentation is produced with a target audience in mind. This target audience will change as the software moves from design to implementation and on to maintenance. All but the maintainers of the software product are a transient audience for the documentation. Maintainers take final delivery of the documentation as part of the software product and must work with it for a long period of time. Keeping the documentation accurate during software development will support maintainers by giving them effective documentation and will allow one set of documentation to serve the design, implementation and maintenance of a software system [66]. Also, by using the same documentation for each step in the process less documentation is produced and its repeated use promotes the discovery and motivates the removal of errors (as was described above).
6.2 Sources of Documentation

Software documentation is not only produced during the development of a software product but also before development starts and during its maintenance. This section describes the sources of information and how they can be documented to make the software product more maintainable.

6.2.1 Pre-Development

This section describes facets of a software product that must be defined before the development begins because they will be used during both development and maintenance.

6.2.1.1 Standards

6.2.1.1.1 Purpose Standards provide a norm for the appearance and the content of a software product.

6.2.1.1.2 Content The standards used to develop a software product should be those used throughout the organisation and must include standards to be used for documentation as well as for programming. Typical sections in the standards manual should include:

 Standards for Document Form These standards describe what the documents should look like, notation used and all other standards to do with the documentation’s form.

 Standards for Document Structure These standards describe how the documents should be structured and what document parts should be included in each document.
Standards for Document Content  These standards describe what information each document part should contain by describing each of the simplest documentation parts.

Programming Standards  Programming standards describe the programming formalisms programmers should adhere to when writing programs. An important goal of this document are standards that help produce code constructs that are resistant to errors introduced during modifications.

6.2.1.1.3 Examples  The case study provides an extensive example of this type of document.

6.2.1.1.4 Usefulness in Maintenance  Standards are the primary means for making the product “Predictable” in the sense described in Section 5.6 so that maintainers find what is being looked for where it is expected to be. Programming standards go on to encourage both program constructs that are robust to modifications as well as programming constructs that provide conventions that cater to programmer’s expectations about what is implemented in the code.

6.2.2 Development

In addition to software the software development process produces a set of documents that describe the software product in detail. In this section the documents that are work products of the development process are described.

6.2.2.1 System Requirements Document

6.2.2.1.1 Purpose  This document defines the real-world problem by describing the external behaviour of the software system as a “black box”. Here the system
is described as a state machine that monitors and controls certain aspects of its environment. [168]

6.2.2.1.2 Content The system requirements must include a description of the environment that identifies a set of quantities that are of concern to users of the system and must associate each of these quantities with a mathematical variable. [135]. The environmental quantities include physical properties (like temperature and currency exchange rates), readings on visible displays, the wishes of the user and even user preferences.

The System Requirements describe exactly the input to the system and how the inputs relate to outputs from the system. The following sections provide a general framework for this document⁴.

The overview provides a description of the system that is being specified and the specification document itself.

The environmental quantities section identifies the environmental quantities to be monitored or controlled⁵ and represents these quantities with mathematical variables. The environment, including existing systems, places constraints on the values of environmental quantities. These restrictions are described by the relation NAT which is defined in the glossary. Beyond NAT the computer system places further constraints on the environmental quantities that are described in the function REQ, also defined in the glossary. Together NAT and REQ are what must be described in the system requirements. [135]

In practical terms this means that each quantity captured by the system and every quantity controlled by the system must be described in detail. In application software the monitored quantities are often captured from the user. Therefore, in

⁴The description has been taken from [66].
⁵These are defined in the glossary

Jens Andexer September 26, 2001
application software the monitored quantities are typically the values captured on
screen, although they may be captured from other sources like files. The controlled
quantities are then the values presented on screen by the system, the features on the
screen like buttons and icons etc., as well as values produced in files, on reports and
where ever else the system produces output.

Typically a System Requirements Document contains:

**Environmental Quantities** Here the monitored and the controlled environmental
quantities are described and each quantity is related to a mathematical function.

**Environmental Conditions** These restrictions placed on system by the environ-
ment not described as part of the monitored and controlled quantities.

**Modes** The operational mode for the system and the transition between the modes.

**Behavioural Requirements** Restrictions placed on the system by the computer
system.

**Expected changes** A description of the changes likely to be required after the
system development has been finished.

### 6.2.2.1.3 Examples

Examples of system requirements documentation can be
found in [64] [167], [168]. The case study provides a detailed example of what part of
a systems requirement document might look like for a large industrial application.

### 6.2.2.1.4 Usefulness in Maintenance

Since the software requirements describe
the external behaviour of the system they are valuable for adaptive and for perfective
maintenance.

Jens Andexer

September 26, 2001
6.2 Sources of Documentation

6.2.2.2 System Design

6.2.2.2.1 Purpose The system design describes how the system solves the problem defined by the system requirements.

6.2.2.2.2 Content In order to document the system design two additional sets of variables must be introduced. The first set of variables represent inputs, the values actually stored as input to the system. The second set of variables represent output, the values produced be the system. The values of both these sets of variables can again be described by time functions.

For an application, the System Design documents the variables the system takes as input and those it produces as output in great detail. It also describes how the environmental quantities described in the System Requirements relate to the input and output variables. This document describes in detail the “meaning” of the input and the output variables in terms of the real-world and it links the problem to the solution.

In addition to the input and output variables the software design is also documented as part of the systems design. The software design describes the technical considerations and design concepts that went into the software design as well as expected changes that were designed for it.

6.2.2.2.3 Examples Examples of system design can be found in [168] and the case study provides a description of what a systems design could look like for a large industrial application.
6.2.2.4 **Usefulness in Maintenance**  This document is particularly useful for perfective maintenance as it describes design concepts that should be included in an enhancement to the system. The expected changes section shows how anticipated enhancements to the system have been encapsulated. This will reduce the effort required to implement these changes and should reduce the rippling effect of these changes through the system [60] because the expected change can be encapsulated.

6.2.2.3 **Software Module Guide**

6.2.2.3.1 **Purpose**  The software module guide states the responsibilities of each module and, by doing so, describes the decomposition of the system into parts. When Object Oriented programming languages are used instead of modules packages and classes can be described here. This document provides an orientation for new staff by showing how functional capabilities are allocated among packages and classes.

6.2.2.3.2 **Content**  The software module guide must document the responsibilities of each module, package and class in the software system in a way that allows the maintainer to understand the modules responsibilities without needing to know details about the internal design. This document gives designers the opportunity to describe the rules used to decompose the software into modules. This is especially important in “fuzzy” areas where additional information, or an unusual application of the design concept, was used in making design decisions [133].

The module guide must contain the following information:

1. The role played by the module in the overall operation of the system [133].

2. The secret associated with the module. It is worth distinguishing between the primary secret, the one being encapsulated, and the secondary secret, the
implementation decisions made by the designer to facilitate the encapsulation [133].

3. The facilities provided by each module, package and class.

4. Any assumptions made, like assuming there is only a two digit year in a date.

6.2.2.3.3 Examples Examples of the software module guide can be found in [16], [133]. The case study provides an example of what part of a module guide for an industrial application could look like.

6.2.2.3.4 Usefulness in Maintenance The module guide is an extremely useful document for software maintenance. This document is ideal when using the “as-needed” strategy of program comprehension. Remembering that the process of comprehension is one of concept assignment⁶ the informal description of the responsibilities of each module or class provides an excellent overview of its functionality, allowing the association of functionality with a “chunk” of the system. This helps to lead maintainers to modules that need modifications when an aspect of the software system needs to be changed.

6.2.2.4 Interface Specification

6.2.2.4.1 Purpose The interface specification documents externally observable behaviour of modules or in Object Oriented languages of packages and of classes.

6.2.2.4.2 Content This document provides a brief description of the interface in prose and a detailed formal description of the behaviour that can be expected by users of its services.

⁶Assigning meaning to a “chunk” of code text.
One possibility, though not the only one\(^7\), for documenting module interface specifications, is the trace assertion method\(^1[140]\) of module interface specification. Invocations of a module’s programs or of a class’ methods produce a history of the demanded behaviour of the module or class. This history can be divided into sequences of invocations called traces. Clearly there is an infinite set of possible traces for a module or class. Within this infinite set of possible traces there are traces that have the same possible future observations or trace *completions*. Traces with the same completion are termed equivalent. By selecting one trace from each set of equivalent traces and making it the representative of the set, making it canonical, the finite set of canonical traces can be documented\(^1[140]\). Notice that the other traces in the set of equivalent traces can be derived from the representative trace\(^8\).

### 6.2.2.4.3 Examples

Examples of documenting module interfaces can be found in \(^1[73], [74], [140], [135]\) and \(^1[43]\). In the case study an example of package and class interface documentation is provided for part of an industrial application.

### 6.2.2.4.4 Usefulness in Maintenance

This document is clearly valuable to maintainers as it documents the externally observable behaviour of a module or class and thereby reveals the functionality offered. The detail of this document allows the maintainer to verify conjectures made while in the comprehension process during inquiry episodes. Used in conjunction with the module guide, the interface specification can help identify modules or classes that could be impacted through a change by describing in detail the data passed between the parts.

### 6.2.2.5 Uses-Relation Document

\(^7\)A survey of these is done in [171]
\(^8\)This is done using induction, *For details please refer to* [43].
6.2 Sources of Documentation

6.2.2.5.1 Purpose The “uses”-relation document describes the relation “uses”, first introduced in Section 2.2.2.1 in the software system. On Object Oriented software each package is made up of classes and each class contains methods. The parts of the software system the “uses” structure relates are the methods, not classes or packages. However, the uses relation document can present the “uses” relationships at a module, package or class level.

6.2.2.5.2 Content The basic data for this document consists of pairs of unambiguous method references. Due to polymorphism, method names are not enough to identify the “used” method. In Object Oriented software the Uses-Relation Document must include the exact target of a message as well as the computer where the target will be found. Because Object Oriented systems are often written for network applications identifying exactly what method is to be used is extremely important.

Due to the difficulty of gathering this type of data and keeping it current the uses relation document is an excellent candidate for automation⁹. A survey of the literature has shown no automated approach that can identify the “uses” relations in software; only the invocation relation is recognised. This is an acceptable compromise when the invocation structure has been closely aligned to the “uses” structure as described in Section 5.3.2.

In its simplest form this data can be presented in two columns, one for the user and the other for the used method. More complex ways of presenting the data are provided by most automated approaches that gather data from the source code. The value of these more complex ways of presenting the “uses” relation data is a subject beyond the scope of this thesis.

⁹The interested reader is referred to [145] [112] for additional reading on this subject
6.2.2.5.3 Examples  Examples of this documentation can be found in [124] and in [121]. The Case Study provides an example of what part of an industrial application might look like.

6.2.2.5.4 Usefulness in Maintenance  The “uses” structure of software provides key information about control dependencies within the system to maintainers. Knowing this structure of the system the maintainer can develop a better impression of the impact of changes on the system and from that can develop a comprehensive unit test strategy.

When unit testing is complete the tested program must be reintroduced into the system. This integration into the existing system must also be tested. Here again the “uses” relation identifies the subset of the system which relies on the changed unit and that must, therefore, be tested during the integration testing.

The documentation of relationships within the software plays a greater role in the maintenance of Object Oriented systems than in systems developed in other models. Object Oriented systems are not easier to understand than other systems unless there is available documentation that fully captures the relationships between classes, class member functions and member variables [31]. With this document the relationships between classes can be fully described.

6.2.2.6 Module Internal Design Document

6.2.2.6.1 Purpose  The module internal design documents the relationship between the externally visible behaviour of a module and its internal data structures and methods. This document serves as a description of the processing that is “hidden” within the module.
6.2.2.6.2 Content Three types of information should be present in this document [135].

1. Data Structure Section - A complete description of the data structure. In non-Object Oriented software these may include variables implemented in “used” modules [135].

2. abstraction function - A mapping from a domain consisting of the program name and data state pairs to a range consisting of canonical traces.

3. LD-relation - The state the program is in when it begins is the starting state and the state it is in when it terminates is the final state. The sequence of states the program goes through when going from its starting state to the terminating state is an execution. An LD-relation is a specification of the behaviour of each program or method in terms of mappings of data states before the programs execution to data states after the programs execution. These mappings can be documented as LD-relations and are described in detail in [136].

6.2.2.6.3 Examples Examples of the module internal design document can be found in [136], [170] and [8]. The Case Study provides an example of what part of an industrial application might look like.

6.2.2.6.4 Usefulness in Maintenance This document can guide maintainers when implementing a change by relating the specification of the modules internal behaviour to the code. Due to the amount and the level of detail in the internal design document maintainers will be tempted to use the code text instead. The display method [136] of program documentation integrates the specification with the implementation thus making the specification more accessible and making it less
tempting to make code changes without first changing the specification. A display consists of the following according to [136] :

- P1 - a specification for the program that is documented in the display.
- P2 - the code text for the program itself.
- P3 - specifications for all programs that are invoked by P2.

By putting the specification and the implementation together in this way the maintainer can easily relate the two. Furthermore, a change needed in the program text can be reflected in the specification without the need to look in additional documents.

6.2.2.7 Lexicon

6.2.2.7.1 Purpose The lexicon's primary purpose is to reduce redundancy in the documentation of a software product.

6.2.2.7.2 Content The lexicon is a central repository for parts that are used more than once in the documentation for a computer system. The lexicon should contain:

- all constants.
- duplicated mathematical functions.
- all names for modules, programs, screens, data bases, etc.
- all error codes and their meanings.
- all database variable names.
- all names that give the maintainer a starting point when confronted with a crash.
The implementation of this document must be in a medium that allows indexing so that the information relevant to a task can be easily extracted. For example, if a single display is being modified then the lexicon entries relevant to that display must be made available in an easily accessible form.

Beyond the extraction requirements the lexicon should serve as a “first stop” point for maintainers confronted with a crash. Too often the information given maintainers to debug these events is extremely vague. Since users view the computer system in terms of its functionality they often will be able to identify what they were doing but not the part of the system they were in. By identifying each screen in the system with a unique identifier a direct mapping between what the user sees and the software is established. By identifying the screen to the maintainer the maintainer can go to the lexicon and find exactly what programs use the screen.

A similar situation is true when the maintainer is “on call”. Being “on call” means the maintainer is responsible for correcting UEs in the software 24 hours a day. Because only systems critical to the organization are supported in this way a UE in one of these systems can be a crisis. The “bug” must be diagnosed as quickly as possible. Once again the information in these situations can be poor but with a lexicon the scant information can identify the parts of the system where further investigation should be focused.

6.2.2.7.3 Examples Examples of the lexicon are in [136]. The Case Study provides an example of what part of an industrial application might look like.

6.2.2.7.4 Usefulness in Maintenance As discussed above the lexicon can be the most important document for the maintainer since it is a central repository\textsuperscript{10} for references that point at places in the system where more investigation should be

\textsuperscript{10}A “one stop shopping” index to the system.
focused for any maintenance task.

6.2.2.8 Design Audit Trail

6.2.2.8.1 Purpose Knowing the history and reasoning behind design decisions can be a valuable asset during perfective maintenance because this provides information about why design alternatives were not used. This document can provide guidance to engineers involved in the next phase of the development process by providing insight into how design decisions were made up to that point.

6.2.2.8.2 Content The design audit trail serves as a repository for the design history of the system, providing information about both why design decisions were accepted as well as why decisions where rejected. The design audit trail is an informal document containing all supporting documentation for design decisions including analyses documentation, studies, preliminary designs, meeting minutes, etc. In spite of its informality, though, it should be possible to refer from the audit trail back to the current documents.

6.2.2.8.3 Examples Case study

6.2.3 Maintenance

The documentation produced during the software maintenance process is used to monitor the software’s operation and modifications made to it.

6.2.3.1 Production Failure Log

6.2.3.1.1 Purpose The production failure log captures information about undesired events and possible solutions.
6.2 Sources of Documentation

6.2.3.1.2 Content The production failure log can be an informal document or a complex data base that captures all relevant information about crashes and about incidents considered unacceptable by users. The captured information should include the following:

1. Date and time the UE occurred.
2. When the UE was detected.
3. Who on the maintenance team responded to the UE.
4. Details of who detected the UE including:
   (a) name.
   (b) phone number.
   (c) job title.
5. If the UE occurred in the batch or on-line system.
6. The program the UE occurred in.
7. As much detail about what was happening in the system when the UE occurred as possible.

Once the initial information has been gathered the log must have a step by step description of each action the responding maintainer takes and the effect the action has.

6.2.3.1.3 Examples Case Study

Jens Andexer

September 26, 2001
6.2.3.1.4 Usefulness in Maintenance  The larger this log gets the more valuable it becomes. Each time a maintainer is called upon to correct a UE in the production system this log is consulted to see if the UE occurred before and what action was take to correct it. This is key to solving most production problems.

If the UE cannot be resolved then the details of its occurrence are retained. An UEs history is invaluable when the job of correcting it becomes a change request.

6.2.3.2 Change Request

6.2.3.2.1 Purpose  The purpose of the CR is to document changes required for a production system.

6.2.3.2.2 Content  This is a working document that describes the original need for change and then logs the progress of the request through the software maintenance process. The change request should begin with a general description of the need for a change to the application system. This description would be primarily provided by users and would be in terms the user could understand.

Content of this document varies but it will always have the following:

1. Identifier - CR identification

2. Owner - Individual responsible for the successful completion of the CR. This individual should have access to both technical as well as user staff and have the authority to make decisions about the priority of the CR.

3. Progress - A log of information about the CRs progress through the software maintenance process with full references to other documents and a record of work done on the CR.

4. User View - A description of the requested change in as much detail as possible in user terms.
5. **Technical View** - A technical description of the change down to a description of the changes to individual modifications in the Software Requirements. Maintainers then receive the modified parts of the Software Requirements documentation.

### 6.2.3.2.3 Examples

Case Study

### 6.2.3.2.4 Usefulness in Maintenance

This document is central to the process of initiating a change to production software. It is a working document that logs the work done on the CR, problems encountered and the progress made.

### 6.2.3.3 Change Request Log

#### 6.2.3.3.1 Purpose

Logs CRs outstanding, those in progress and those completed and provides an index into the CRs. This can be useful during corrective maintenance when tracking down production problems that started when the last set of changes was moved into that environment.

#### 6.2.3.3.2 Content

A change request or CR is a formal request to have a change made to a production system. The change request log should include the following:

1. CR identification.
2. Maintainer(s) responsible for the CR and all specs.
3. Scheduled date for move to the production system.
4. Actual date moved to production system.
5. Identification Identification of the group of items moved to production.
6. CR(s) that produced the changes.
7. Target production libraries.

8. Where what is to be moved can be found.

9. Itemized list of what is to be moved.

10. Programmers responsible for changes.

6.2.3.3.3 Examples Case Study.

6.2.3.3.4 Usefulness in Maintenance Since most production problems are introduced through changes to the production software the value of this document is obvious. If there is an UE in production in a program that was just moved in with a CR then invariably it is the result of the changes. This allows the maintainer responsible for the CR to be called upon for help. This document can also be used to see what is scheduled to be moved in to production in order to anticipate potential problems and for scheduling who will be “on-call” to coincide with what is moved to production.

6.2.3.4 Production Migration Log

6.2.3.4.1 Purpose This document logs the changes that have been migrated to production. The Production Migration Log must contain enough information to allow the re-creation of any version of the software. In some industries it is vital, for legislative reasons and for audit, to be able to recreate a past data state. For example, a health insurance company in Germany sent out annual statements that, in some instances, were not correct. This went unnoticed for some time until one of the insured realised the error and when it first appeared. The insurance company could track down the problem by reviewing the changes that went into production using the Production Migration Log. The extent of the problem could then be determined
and the effected policies returned to their state before the error and recalculated to remove the defect.

6.2.3.4.2 Content  The Production Migration Log must be a precise log of everything that is moved to production and when it was moved. The contents of this document vary but should always include:

1. Identification of the group of items moved to production.

2. Scheduled move date.

3. Actual move date.

4. CR(s) that produced the changes.

5. Target production libraries.

6. Staging libraries (Libraries where software is kept before it is moved to production).

7. Itemized list of what is to be moved from where it comes and where it goes (everything that is to be moved!).

8. Programmers responsible for the changes.

9. How the programmers can be contacted if something goes wrong.

6.2.3.4.3 Examples Case Study.
6.2.3.4.4 Usefulness in Maintenance  This log is the source of information about what changed in production and when it changed. During corrective maintenance when UEs start to be detected that were not evident before this log can provide an invaluable source of information for where to start looking for new production problems.

6.2.3.5 In Code Documentation

6.2.3.5.1 Purpose  This form of documentation describes implementation details and logs changes to the code within the code text itself.

6.2.3.5.2 Content  In code documentation should be included in the programs by developers. It is vital to maintainers that this documentation be kept current. In code documentation should only add to information found in other documents and should not duplicate it. Because the documentation is within the code the integrity of the duplication of information is difficult to keep. Therefore, In code documentation should point the maintenance programmer to other sources of information rather that put this into the code text.

The documentation should consists of:

1. Comments at start of program. These comments give a general description of the program and draw attention to program design concepts and implementation details the developer considered important.

2. Program change history. The change history starts with an entry for the creation of the program, as it was when it first entered production. Then there must be an entry for each change made to the program during its time in production. This information must include:
6.3 Conclusion

- Identification number for changed item. A sequential number to identify the change.
- Date changes where started and finished.
- Programmer making changes.
- CR that initiated changes.
- Brief description of changes.

3. In line documentation of logic. Tricky implementation issues are documented here.

4. In line documentation of changes. Identification of where code was removed, added or modified. Each change should be identified with the identification number charge history found at the top of the program.

6.2.3.5.3 Examples Case Study.

6.2.3.5.4 Usefulness in Maintenance In code documentation is vitally important for maintenance programmers. This is an excellent place to describe subtle but important implementation issues. The in code documentation should also describe how modifications to the program were implemented. This can help the maintainer track down problems in the production system introduced by maintenance activities.

6.3 Conclusion

A large amount of detailed information about how an organisation carries out its business resides in its software. This information is in the form of programs that carry out business processes. Although volumes may exist describing how the business is meant to be done, how it is really carried out is determined by its programming.
Users of the software expect it to react in a predictable manner and rely on this predictable behaviour to carry out their work. They have little knowledge, however, of the information hidden in the automated processes they use. Documentation can make much of the hidden information accessible.

When new software is written or when existing software is changed programmers need a detailed understanding of how programs implement business processes. In the absence of documentation this information can only come from the people and from programs. A dialogue must take place in which detailed information about what the programs do is communicated to users and information about what the software should do is communicated to the programmers. This dialogue almost always has the effect that the organisation redisCOVERs information about itself and how it carries out its business. This is a rediscovery because, as described in Section 4.3, the detailed information in the programs is extremely difficulty to retrieve. Programmes are a poor medium for retaining and communicating information [2] [166] while the documentation described in this section has been designed to be an effective communications medium.

In Section 4.3 one of the key sources of information about computer systems was described as other programmers. However, Yip in [178] identifies staff turn over as a key problem for maintenance departments. Staff turnover confronts maintenance departments with the situation where knowledge about the software is drained by staff leaving the organisation. This makes maintainers themselves a poor medium for retaining information about the software. The documentation presented provides an effective, easily accessed medium for storing information about the software that cannot be drained away by staffing problems.

This chapter introduces a set of documentation that will provide a detailed description of a software system in a concise, easily accessed form. In fact, the infor-
6.3 Conclusion

Information available from good documentation is more valuable than the working code. Accurate documentation can be used to produce new software while changing an undocumented system will result in its degradation to a state where the software must be redeveloped due to high maintenance costs\textsuperscript{11}.

\textsuperscript{11}This was discussed in Section 2.2.
Chapter 7

Case Study

The purpose of the case study is to show how a large software system can be documented so that the documentation supports maintenance and to go on to show how that documentation does support maintenance programmers.

The case study shows the key role documentation has on software maintainability by providing a cross section through a set of documentation that has been developed for maintenance.

An important characteristic of software maintenance is that, typically, software maintenance is “programming-in-the-large”[34]. It would, therefore, be of little value to the reader if this case study were to show how a small, tidy software system can be completely documented. The software systems found in industry are neither small nor are they tidy. Therefore, this case study shows the partial documentation of a large system and goes on to show how this documentation supports typical maintenance activities.

An insurance product has been selected for the case study because the financial services sector relies heavily on computer systems and has done so for many years. The computer systems installed long ago are often still in use and will continue to be used for some time. As a result the financial sector has led the way in accepting and

Jens Andexer

September 26, 2001
integrating new technologies into their organisations while maintaining software for long periods.

Problems arising from mixing old and new technologies are made acute in a volatile market place. Financial services are frequently affected by changes in the market place stemming from new regulatory legislation. These changes often require existing financial products to be modified or provide the opportunity for new offerings. For example, the introduction by LAUTRO\(^1\) of standardised quotations for the marketing of personal pensions in the UK during the mid 1980s led to significant changes in the quotations software for personal pensions. Similarly the introduction of Registered Retirement Savings Plans in Canada led to the introduction of new financial products here. Both of these regulatory changes required extensive changes to existing software systems or required the development of new software systems that support new products.

In this case study a fictitious firm, A\(_K\)E Insurance, has a product for the home owner that provides for some customers to insure multiple properties with various uses in different countries. This is called the Home Owner Freedom product.

The first section in this chapter explains key features of the case study, the second section describes typical maintenance activities and how the documentation supports these. Finally, observations made while the case study was being developed are presented.

### 7.1 Key Points in the Case Study

This section will describe how maintenance tasks described in Chapter 3 can be made easier through documentation paradigms presented in Chapter 6.

\(^{1}\)LAUTRO, Life Assurance and Unit Trust Regulatory Organisation, is responsible primarily for regulating the marketing of financial products of member companies in the UK.
7.1 Key Points in the Case Study

7.1.1 Reuse in Documentation

By reusing parts of the documentation complex modifications can be made easier.

7.1.1.1 VBB

In the case study a reusable documentation component has been formulated. Called the Visual Building Block (VBB), this documentation form allows a detailed, precise description of some aspect of the software system. When reused elsewhere, the VBB provides the same detail and precision but without redundancy. This makes the VBB convenient for describing screen interactions that are used more than once.

For example the text1_vbb, text30_vbb and the address_vbb have been developed to show how a simple VBB can be used, but also to show how the VBB encapsulates complexity and keeps changes local. If, for example, a change were introduced to enhance test1_vbb this would only need to be documented once, in text1_vbb. Other documented parts, like g1hf325e_pl, would stay unchanged.

7.1.1.2 The Lexicon

The lexicon is another device the case study uses to provide reusable documentation parts. Here reusable parts like complex functions, expressions, constants and images are documented once for reuse in the rest of the documentation.

7.1.2 Simplicity

Throughout the case study very simple mathematics have been used and where this was not possible expressions have been given descriptive names and defined in the Lexicon. This makes the documentation read more like the technical texts the program’s maintainers are accustomed to reading while still providing the detail when it is required.

Jens Andexer

September 26, 2001
7.1.3 Standards

A standards Manual has been developed that provides enough information about the structure, content and format of the documents to be useful without overwhelming the reader with detail. To this end the manual has been broken into two parts.

7.1.3.1 Documentation standards

The goal of the documentation standards is to produce documentation that is easy to use and easy to change and that has a form that will not deteriorate as the documentation is maintained. This has been done by structuring the documentation as shown in the standards for document content. By describing the contents of documents in detail and by standardising how the documents should be structured their deterioration is slowed.

7.1.3.2 Programming standards

The goal of the programming standards is to guide programmers to produce programs that will still be maintainable after they have been under maintenance for a long time. Change logs at the top of each method provide the programmer with a way of finding where changes have been made in the code text. These are a useful guide since changes are an important source of errors.

The “$>$start” and “$>$end” are used to identify where the non-declarative part of the code begins and ends. This may seem redundant at the beginning of a methods time under maintenance. After fifteen years and potentially several dozen changes, each change adding code and declarations, the resulting code can bloat to an unforeseen size. When the method has become a leviathan, knowing where it starts and ends can be a significant benefit.

Finally, control structures have been defined that are resistant to new errors.
This is done by insisting on the beginning and ending of a control block always being explicitly coded. The case study standards insist on opening and closing braces for all control structures. It is possible to avoid coding these if the code behind the condition or in the loop is only one line long. However, if another line must be added then adding the needed braces may be forgotten.

7.1.3.3 The Compliance of Standards with Conventions

What the standards manual could not show is where the standards need to conform to existing conventions. The “normative power of the factual”\(^2\) is a key issue when introducing standards to an existing set of procedures. Therefore, the case study uses the same capitalization standards for Java\(^{TM}\) naming as it the industry norm. As noted in the case study, what standards are applied is less important than how consistently they are applied.

7.1.4 The Index

An index has been created that cross references where each documented part is defined and where it is referred to. This allows a programmer to identify the parts of the documentation required for an as-needed analysis\(^3\). In addition to identifying the page where the entity is used, the index identifies the document the reference will be found in. The document reference is provided using the document abbreviation found at the top of each page and listed in the standards manual.

Beyond its uses as a cross reference the index also helps to identify parts of the documentation that are defined but never used or used but not defined. This is done by showing where the element is defined in dark type and where it is used in a normal type face.

\(^2\)“der normativen Kraft des Faktischen” is translated here from a text by Georg Jellinek in [75]

\(^3\)This will be shown in Section 7.2.

Jens Andexer  September 26, 2001
7.1.5 Cross Referencing Information

The case study uses two means of tagging information for cross referencing, each with a specific purpose.

7.1.5.1 Mnemonic Identifier

Mnemonic identifiers can act as triggers for a programmer’s long term memory. To do so the identifier must be sufficiently descriptive to have this triggering effect.

The case study has a descriptive identifier associated with each documented part. These identifiers are used in the documentation and throughout the software, allowing easier association of functionality in the software to its documentation.

Reusing the identifying mnemonic in this way has another advantage. Information is moved from short term to long term memory easily but the association between the trigger and long term memory must be built up through rehearsal. Rehearsal is the process of building associations between long term memory and short term memory. The more a concept is rehearsed, the more likely it is to recalled[62]. By reusing the mnemonics the concept associated with them is reinforced and its correct recall more likely.

7.1.5.2 Unique reference

Everything documented in the case study is provided with an Entity Reference, described in the Standards for Document Appearance, that appears against the left hand page margin. For example, the controlled quantity defining the x coordinate for the top left corner of the g1hf325e panel is cq-1.1. This key uniquely identifies this piece of information. Indeed, cq-1.1-2 is the text description of this controlled quantity. cq-1.1-2 identifies this description uniquely.

Entity References are only associated with one entity and cannot be reused when
the entity is dropped from the system. This makes the identifier not only unique to the current version of the system but to all versions. Reconciling different versions of the same documentation is, therefore, manageable. Also, by not allowing the identifier to be reused the same name can be used for different entities. For example, a monitored quantity and a controlled quantity could both be called “riskCountry”. The identifier allows the two quantities to be differentiated and still uniquely identified.

This unique identifier is invaluable when a specific detail in the documentation must be referred to. Often maintainers that speak different languages must communicate detailed information about the system under maintenance to one another while being geographically separated. Without the Entity Reference it is difficult to ensure all parties are focused on the same details. With multiple versions of the same documentation and software distributed and maintained from different geographic regions ambiguity about details can lead to a serious waste of effort and to serious errors.

7.1.5.3 Design Audit Trail

Often it is as important to know why an aspect of the software design was not included as it is to know the software design. The objective of the design audit trail is to provide a link from the diverse technical information that confronts a designer at the beginning of a development to the formalism of the software design. In order to provide this link, the document must give the designer a forum in which non-functional features of the system can be noted but that also allows the key points to be referenced. The Case Study does this by making the Design Audit Trail an informal document with references to key points. These references are then used later when the rational for breaking the system into modules is documented.
7.1.5.4 Non-Functional Considerations

This section of the Design Decisions provides a general overview of many of the technical issues that must be considered before developing a new application and integrating it into existing systems. The section is written in the general imprecise language typical of this type of document. Its purpose is to hold fast the merits of technical solutions that were not successful. It also suggests the reason why these solutions were dismissed. Often solutions that look very promising on the surface have one fatal flaw that makes them unacceptable, as is the case for alternative one. There is no reason not to use traditional technologies other than access for users across the Internet. Traditional technologies cannot provide a user interface acceptable to Internet users and since this type of access is deemed important Alternative 1 cannot be used.

7.2 How the Documentation Supports Maintenance

In this section two typical software maintenance tasks are described in order to show how documentation can make maintenance work more effective.

7.2.1 Adaptive and Perfective Maintenance

For the maintainer, Perfective and Adaptive maintenance begins with a change request. The change request CR-236 can be found in the change request documentation section of the case study. This CR describes that the Home Owner Freedom product needs to be extended to include a new validation to the “Risk Country” input data and goes on to describe that this data is captured on the “Risk” panels. The impact analysis identifies the RiskCountry class as being impacted by the change but does not show the analysis that led to this conclusion. This section will describe how a
maintainer might have come to the conclusions documented in the CR.\textsuperscript{4}

The panels on which the data is entered are identified in CR-236 as being all those beginning with G1HF325. This can be derived from standard pX-1-1.1 which is the naming convention for panels described in standards documentation. In the Software Requirements the panel definition for G1HF325E shows that the “Risk Country” literal on screen as associated with a field called \texttt{riskCountry}_{mq} which is at coordinates $x^3_T$, $y^6_L$. This Monitored quantity can be identified with the reference \texttt{mq-32.1}. The monitored quantity \texttt{riskCountry}_{mq}, reference \texttt{mq-32.1}, is then used to define the input variable \texttt{hfriskCountry}_{iv} which has the reference \texttt{iv-2}. This association can be made through the index where the definition of the monitored quantity can be found along with where it is used.

Having found \texttt{riskCountry}_{iv} as an input variable the data entry panels that capture the risk can be identified through the monitored quantities used to define the input variable. These are the panels that must display the error message for the new validation. By looking at the panel layout the maintainer can identify the error message field; it is the VBB at the bottom of each screen called \texttt{messageToUser}_{obb}. This information can be found by analysing one of the panel definitions, by knowing this is the message field from experience or by asking someone. All sources of information are about equally effective.

Once the \texttt{messageToUser}_{obb} has been identified as where the result of a failed validation should be reported it follows that this output variable needs modification. \texttt{messageToUser}_{ov} must be changed to allow the newly detected incident to be reported. Therefore, error message code 523 has been added to the output variable and to the Lexicon. This facilitates the newly detected invalid data to be reported to the user in either English or German. The message and the expression to detect the UE,

\textsuperscript{4}For an explanation of the references and the meaning of subscripts used in this section the reader is referred to the standards manual in the case study.
however, are defined in the lexicon since these must be reusable.

During the implementation of this CR the programmer must know that the \texttt{aceGUI} package implements the needed behaviour. This is not an unreasonable deduction since all user interface objects are defined in this package but the information is also provided in the module guide for reference ml-2-1.1. The package interface documentation would then be used to confirm \texttt{aceGUI} as the package providing the needed behaviour. From here the ic-17.6-2 identifies the \texttt{CountrySln}\textsubscript{5} class as implementing the necessary behaviour.

Now the code change can be carried out and testing can begin. The code changes must be implemented in two distinct steps. First, the validation must be added and this change moved to production. At this point the validation will always result in true because it checks against the countries in the array in the drop down box. By putting the validation in first it stops errors from being introduced into the data. If CR-239 is mistakenly put into production before the validation is complete then CR-236 will report an error when a user keys in a country not in the displayed list.

### 7.2.2 Corrective Maintenance

The case study provides two failure scenarios to show how documentation can be used to support programmers when confronted with this type of problem.

#### 7.2.2.1 The Program’s Responses to Detected UEs

UEs are detected and handled during the execution of a program’s code. However, which UEs are detected and the programs response is described in the program specification. The problem for maintainers is often the association of the programs response to a UE when it is detected and the description of that response in the specification.

\footnote{Not included in the Case Study.}
To help programmers in this association each UE detected in the software is identified by the UE Detection Reference, programming standard pX-1-3, which is an identifier that is unique throughout the system. UE Detection Reference is documented in the specification and also in the code text where the code detecting a specific UE is located. A programmer needs only to scan the code text and documentation for the UE Detection Reference to find out where the UE is detected and where its detection is documented.

7.2.2.2 The Human Response to Detected UEs

The Production Failure Log provides two scenarios typical of that type of document. The first is, on the surface, a trivial problem involving browser settings. However, the user cannot carry on with their work and this escalates the importance of the problem significantly. The solution to this problem is easily found using the Failure Log. Some one else had already gone through the trouble of diagnosing what could be a tricky problem and documenting a solution. There is no need to diagnose the problem again; a process that could result in reading code to see where JavaScript is being initiated and why it is not happening correctly in this case if the information is already available.

The second scenario involves a misinterpretation of user requirements coupled with an error with the migration of a change that results in a design flaw being introduced into the system. This, also a potentially tricky problem, can be diagnosed using the documentation. The scenario also shows the importance of the entity reference. The programmer dealing with the error and the programmer who made the change are geographically separated and can communicate only by telephone. Still, by using the reference of the section in question, they can immediately begin discussion about the problem without ambiguity about which section is meant. In a small CR like CR-236 this may seem superfluous, however, in large complicated change requests this

Jens Andexer

September 26, 2001
ambiguity becomes a serious source of errors.

7.3 Observations Made during the case study

This section describes some observations made while the case study was developed and provides insight into strengths and weaknesses found during implementation of the guidelines espoused in the thesis.

7.3.1 Document formatting

It became obvious early on in the development of the documentation for the case study that the text formatting software was not up to the job. Although it produced a professional looking product the formality needed for the documents in this thesis required documentation constructs that were time consuming and difficult to develop while providing only a partial fulfillment of the needs. As a consequence the structures that were developed created a source document text that looks significantly different from the final output documents. This makes the association of the two difficult, time consuming and error prone.

7.3.2 Document Structure that is Fixed by the Medium

A limitation of the documentation that became evident while developing the case study were the structures of the documents themselves. In the same way that software has structures its documentation also has structures that conform to relationships between the parts being documented. In the case study the most important relations can be described as “is available from” and “is defined in terms of”.

The structures in the case study have been developed to group information in a way convenient for developing this case study. This arrangement, however, is not ideal for some maintenance tasks. For example if Ace were purchased by another
company and the business of capturing risk data changed then a large part of the Home Owner Freedom system documentation would be affected.

No matter how the case study documentation is arranged some maintenance activities will delocalise and become more difficult as a consequence. The reason for the delocalisation is an unfortunate grouping of information. What is convenient for one task may be inconvenient for another. The problem is that the relations between document parts are frozen when the document is printed, an effect of the medium the case study is written in. Paper is a medium that is difficult to change once information has been committed to it. As a result the case study is structured to make some tasks easy with the inevitable effect of making other tasks more difficult.

What would help maintainers is a way to slice\textsuperscript{6} the documentation so that only those parts of the documentation necessary to perform a specific maintenance task are visible. This type of behaviour in the documentation will require a departure from documentation whose structures are fixed. By putting the sum of the documentation into a database, the readable forms of the documentation become views\textsuperscript{7} of the database. These views would then be slices of the documentation that provide information relevant to a given maintenance task.

### 7.3.3 Inadequacies of Documentation Media

Beyond the disadvantage of paper described above is the drawback that many of the images used on screen can have, a behaviour that is not part of the functionality of the system being documented. The Ace Icon, for example, may in fact be a gif\textsuperscript{8} with complex but purely visual characteristics. This gif may not be documented but is still desirable for inclusion in the documentation of the Home Owner Freedom system to

---

\textsuperscript{6}This term is defined in the glossary.

\textsuperscript{7}Program slicing is a technique described in [172] in which programes are analysed and reduced to produce only a desired set of behaviour. Here the same idea is applied to documentation.

\textsuperscript{8}Graphical Interface File

---

Jens Andexer

September 26, 2001
show its behaviour. On paper, however, the complex non-functional visual behaviour provided by some graphical elements, like gifs, or audio behaviour, like real audio files, can only be documented with a significant amount of effort that brings little advantage.

7.4 Conclusion

Bringing theory closer to practice is never an easy task and the case study has been no exception. What the case study shows is that a set of documentation can be developed that is sufficiently precise and easy to use to make it in the programmers best personal interest to keep documentation current. It would be in the programmers best interest because it makes maintenance tasks easier and less error prone. Therefore, the documentation should have “grass roots” support. If maintenance programmers can be more effective in their job and, therefore, save money by using the documentation, this type of documentation should also receive management support.
Chapter 8

Conclusion

This chapter discusses the potential applications of the design guidelines and documentation paradigms for Object Oriented programs along with some of its limitations.

8.1 Applications for This Work

In contrast to most of the existing literature on designing software this thesis clearly identifies factors that contribute to increasing software maintenance effort. By first analysing the difficulties encountered during software maintenance and then using these to identify design guidelines that support the software maintenance process this thesis connects design decisions made during development with the effect those decisions have on the maintainability of the final software product. This reveals an important connection between decisions made during the development of a software product and the long term implications of those decisions.

The thesis goes on to identify documentation, the medium for communicating design, as fundamental to reducing maintenance costs. The net reduction in maintenance costs brought about by documentation produces the paradox of cost cutting at the expense of documentation. As documentation is neglected it becomes a less effective medium for communicating details about the software design. The result
of the neglect is that programmers resort to the code text for information about the software behaviour. The benefits realized by removing the documentation overhead are more than offset by the burden of increased effort during the software change process.

8.2 Limitations

Users of this paper should be aware of the limitations of the ideas presented herein. Adhering to the design guidelines and documentation paradigms will result in systems that are more maintainable and, therefore, less costly to change. It is, however, not possible to determine the maintainability of software other than by subjective means. A system is considered more maintainable because it is easy to change but it is not possible to determine if system A is easier to maintain than system B. Even the notion that system A is less costly to maintain than system B is not meaningful. How to measure such aspects of software is still too poorly understood.

Users of documentation in a maintenance environment need a way of conceptualizing how the software is interconnected in order to understand how functionality maps to implementation. Mathematics is a very precise and unambiguous way of documenting requirements and specifying software behaviour. Unfortunately most maintainers find it difficult to conceptualize software structures from mathematical specifications.

8.3 Future Work

Maintaining software is expensive. As a result, currently the most used measure of maintainability is cost. Although the primary factor contributing to cost is known to be effort, how to reduce the effort is only superficially understood. Many characteristics software should have in order to make it more maintainable have been described
in this thesis. But how much more maintainable these factors make the software is not known; maintainability can still only be subjectively determined. What is needed is a way of identifying and measuring key aspects of the software that affect maintainability so that software maintainability becomes less subjective; only when aspects of maintainability can be measured can costs be assigned to them. When software costs can be correctly assigned they can be intelligently controlled.

The benefits of mathematics as a specification language for software are not disputed. What is disputed is the value of mathematics as a tool for conceptualizing a software system. Work is underway in the Table Tool System (TTS)[143] to help in this respect but more work is needed in the conceptualization of software, particularly when mathematical specifications are used.

Finally, the conceptualization of software is a difficult task. Building hierarchical structure into the “uses” relation is not easy because the dependencies that make up the uses relation can be difficult to detect. The source of the difficulty comes from complexity, an essential property of software [52]. When confronted with several hundred modules and several thousand relationships between modules it is difficult to determine how adding a new relationship will compromise the software design.

Tools can play an important role in analysing software artifacts to support software maintenance. What is clear in Figure 2.1 and from the absence of tools in software maintenance practice is that the maintainer’s needs are not being met. Singer et al. in [158] identify some key needs maintainers have for tools that continue to go unanswered.

What is needed are solutions that integrate a programs documentation with the code. Such solutions would allow programmers to filter out documentation they do not need while helping them find the documentation they do need, thus helping in an as-needed analysis of documentation. It would be desirable to have this facility in
the form of a flexible query language that allows maintainers to ask questions of the documentation and interactively narrow the scope to the needed parts. In addition support is needed when changing focus from documentation to program code. Once the needed documentation has been identified the program code implementing this documentation must be easily identifiable.

A scenario like that described above could also be made to support the automated documentation of existing systems. The code body, in this case, could be analysed and the result put into the documentation system described above. This should not replace the accurate documentation of a system from the beginning but would prove instructive when comparing the existing documentation with that produced by automated means. The above solution could also extend the time existing systems stay in production; systems that are currently maintained without the benefit of accurate documentation.

8.4 Conclusion

Although there are indisputable benefits to using computers, Strassmann in [163] presents data showing that in 55% of US Firms the Information Technologies (IT) budget exceeds the economic value added to the firm by its investment in IT. Strassmann goes on to argue that no demonstrable link exists between IT spending and profits. If IT costs represented only a small portion of corporate spending they would be relatively unimportant for profitability. But IT spending is significant in most firms\(^1\) and the largest single contributor to IT costs is software maintenance[159].

Unfortunately software is still poorly understood. Decisions affecting software often have long periods between when the decision is made and when the effect of the

---

\(^1\)US Corporate spending on IT was USD 500 billion in 1995, exceeding the sum of 1995 corporate profits by USD 175 billion. [179].
8.4 Conclusion

decision can be recognised. The disconnection between cause and effect results in a
great deal of uncertainty in the decision making processes for software development
and maintenance. Decisions aimed at cost reduction are usually thoughtful decisions
made by competent people using the best information available. Unfortunately, due
to the difficulty in attributing effect to cause the best information is often poor and
difficult to interpret. The result is decisions that frequently favour more easily justified
short term gains at the expense of the long term. This tendency toward the short
term makes it appear that IT costs are difficult to control.

The primary source of costs during software maintenance is clear. Maintenance
costs arise from the effort needed to correctly make a change to an existing software
product. Less clear are the factors that contribute to the effort required to make
changes during software maintenance. In an attempt to regain control over the effort
needed to make changes to software, industry is looking to technological innovation
for solutions; the Object Oriented paradigm is such an innovation.

If Object Oriented languages are used to develop software and due effort is spent
on the design and documentation of the software products then maintenance effort
can be reduced. If, however, the decision is made to save in the short term by
not designing and documenting the software adequately then using Object Oriented
technologies can result in increased costs. The Object Oriented paradigm is not a
panacea\(^2\) and when abused it has been shown that it can result in less maintainable
software and in faster degradation than software developed using traditional models.
In Object Oriented software, according to [30] [31] [69], the design and documentation
is even more important to maintainability than in the procedural paradigm.

It is difficult to improve when there is no way of measuring progress. Currently
industry seldom measures software other than through costs\(^80\). Cost alone is hope-

\(^2\)As Brooks observes in [52] the Object Oriented paradigm is no “silver bullet”.
lessly inadequate as a meaningful measure for progress toward software maintainability\(^3\). This situation is worsened by an inability in industry to accurately account for costs incurred or saved by characteristics of a software product. Developers are rewarded for finishing software within a redefined schedule and budget while there is little regard for the quality of the final product. The work lost due to poor software is also seldom accounted for. The full cost of poor software in terms of increased maintenance effort and lost user effort can only be guessed at. Until the benefit of design decisions and documentation produced during development can be clearly assigned to cost saving during maintenance there will be great reluctance to incur costs in these areas. Although cost reduction must stay the long term goal better ways are needed to measure costs as well as to measure progress towards that goal of cost reduction\(^4\).

This thesis has shown that effort spent on software design and documentation can reduce the long term effort needed for software maintenance. Currently this reduction in effort cannot be measured. To make the case in industry that it is worth carrying the front end costs to software development for a benefit that is only realised after implementation, the benefit must become measurable. To make the benefit measurable software in general and software maintenance in specific must become better understood.

\(^3\)Foster in[50] argues that improvements in the maintenance process must be measured by extensions in average program life since maintenance costs are kept a fixed proportion of an organisation’s overall expenditure.

\(^4\)Some better accounting practices for assigning maintenance costs are developed in [51].
Chapter 9

Standards used in this Thesis

9.1 Notation

The following typefaces will be used to identify names.

System names are written as System

Package names are write as Package

Class names are write as Class

Method names are write as Method

Variable names are write as Variable

9.2 Index Notation

Page numbers for index entries can have one of the following three type faces.

normal type face page were the term is used in the text.

bold type face page where the term is defined in the body of the text.

italic type face page where the term is defined.
9.3 Glossary of Acronyms

A

A7E A7E Aircraft

API Application Programming Interface

CR Change Request

DBMS Data Base Management System

FIFO First In First Out

IN Relation IN, For details please refer to IN in the Glossary of Terms.

IT Information Technologies.

JCL Job Control Language: a language to control the execution of programs on mainframe computers.

JRE Java\textsuperscript{TM} Run-time Environment

JVM Java\textsuperscript{TM} Virtual Machine

Jens Andeexer

September 26, 2001
LIFO  Last In First Out

LXC  Lines of Executable Code (No comments) where a line of code is a syntactically complete program statement, even if it goes over more than one line of code text.

LOC  Lines of Code (Including comments). Typically this is measured by the number of records in the source file text.

NAT  Relation NAT, For details please refer to NAT in the Glossary of Terms.

NC  Network Computer

OUT  Relation OUT, For details please refer to OUT in the Glossary of Terms.

REQ  Relation REQ, For details please refer to REQ in the Glossary of Terms.

UE  undesired event

Y2K  Year 2000
9.4 Glossary of Terms

**abstract class** A class that can not itself be instanciated, it can only be extended.

**abstract data type** A class of objects defined by a representation-independent\(^1\) specification[59]. An abstract data type therefore hides, makes a secret of\(^2\), how the data is represented, how it is implemented.

**adaptive** maintenance Modification of software to accommodate changes in its technical hardware and software environment.

**Application Programming Interface** All variables and behaviour of a method, class, or package that are externally visible. In Java\(^TM\) this includes exceptions.

**base machine** The machine on which a virtual machine is based. A base machine can be an interface to hardware, an interface to external software or an lower level of abstraction in the current design.

**call stack** see invocation stack

**change request** Request to change the software based on change in requirements.

**class** Definition of the properties and method’s objects that are instanciates of that class will have.

---

\(^1\)See “type”.

\(^2\)see Section 5.1
Definition 9.1 (Class)
“A class is a set of objects that share a common structure and a common behaviour. [12]”

code text The text of the source code of a program.

coding standards Rules for the coding of programs. These typically lay down how indentation is to be handled, what in-code documentation should be supplied, what programming constructs may or may not be used, etc.

cohesion For details please refer to Section 5.1 on page 101.

compile Translation of source code text into machine code.

component A set of related modules.

computer system All the hardware and software associated with the solution to a real world problem with a digital computer.

control dependency A dependency within software that results from the number of possible paths of control through the program[113]

containment A relation between classes that is not Inheritance. A containment association between classes occurs when a class uses another class as a type.

controlled quantities Environmental quantities whose values a computer system is to control [134].
Definition 9.2 (Controlled Quantities)

Environmental values the system is intended to restrict.

"Let \( p \) be the number of quantities to be controlled in system \( S \), and \( F_C \) a selected set of time-functions (this set is determined by the physical characteristics of the system \( S \)). Then:

1. controlled quantities will be represented by a \( p \)-tuple called a controlled state function, \( c = (c_1, \ldots, c_p) \), where each \( c_i \) is a member of \( F_C \) and corresponds to the \( i^{th} \) controlled quantity; and
2. the set of controlled state functions will be denoted by \( C_S \). "

\[ 3 \] [168]

corrective maintenance The diagnosis and correction of errors in the software.
correct If software meets its specification it is correct. [125].
coupling For details please refer to Section 5.2 on page 107.
crash Undesired events that are not expected and for which recovery is not successful.
change request Request for a software change that initiates the software maintenance process.

\[ \text{D} \]

data Any representation of characters or analogue quantities to which meaning might be assigned. [76]

\[ 3 \] For simplicity it will be assumed that the system under consideration is system \( S \) allowing the omission of the subscript \( S \) in some of the denotations [168].
9.4 Glossary of Terms

“Representation of a precise, formalised language of some facts of concepts, often numeric or alphabetic values, in a manner which can be manipulated by a computational method.” [82]

data base management system Software dedicated to the storage and retrieval of data on a data base

data dependency A dependency within software that results from relationships in the data like shared data references or data that is conceptually related but separately stored

data space All possible data values for all variables in the program, or the union of all domains of all instructions in a program [108]

\[ D = \bigcup \{ domain (z_i) \} \quad i = 0, 1, \ldots \]

where

\[ z \in I \]

\[ I = \{ SetOfInstructions \} \]

data value Member of data space

definitional dependency A dependency between data and how it is defined in a program or sub-program and how it is defined in a program

declaration A source language instruction used to associate an identifier with what it is identifying. For example a type can be associated with an identifier so that when storage is instanciated the object can be referenced. A method must also be declared but here the method's name is associated with its signature.

deliverables The products of the development process such as software and documentation
Dialogue Management System  Software dedicated to managing the dialogue with the video display devices.

discrepancy report  Report about problems or anomalies discovered in operational software.

entry point  A point in a program from where control can be passed to another program, i.e., where another program can be invoked.

entropy  The process of a system moving to a state of disorder due to change.

environment  The problem domain or the part of the real-world a computer system must operate in. A computer systems interactions with the environment is through monitored quantities and controlled quantities.

environmental quantities  Quantities through which the computer system interacts with the problem domain. Environmental quantities exist in nature\textsuperscript{4} and include physical properties such as temperature and pressure, the readings on user visible displays\cite{135} and values supplied by users as input. It is useful to characterize each environmental quantity as either monitored quantities or controlled quantities.

error  Software behaviour that is unacceptable.

event  A occurrence which is independent of time. For instance the invocation of one program by another is an event but does not rely on time to occur.

execution  A sequence of state values \cite{108}

\[ S_i = (d_i, f_i), i = 0, 1, 2, \ldots \]

\textsuperscript{4}see quantity
such that
\[(d_i, s_i + 1) \in f_i, \; i = 0, 1, 2 \ldots\]

which terminates, if ever, when \(d_i \notin \text{domain } f_i\)

where \((x, y) \in f\) or \(f(x) = y\)

\(d_i\) is a data value

\(s_i\) is a state value

**external entry point** Entry point where the target reference is external to the scope of the current program. So for the reference to be resolved requires a search of object libraries.

**formal methods** The use of mathematical techniques from logic and discrete mathematics in the specification, design and construction of computer systems and software. The word “Formal” in formal methods derives from “to do with form” [118].

**finite state machine** A abstract model of a machine with primitive internal memory. A finite state machine \(M\) consists of

a) a finite set \(I\) of input symbols.

b) a finite set \(O\) of output symbols.

c) a finite set \(S\) of states.

d) a next state function \(f\) from \(S \times L\) into \(S\).

e) an output function \(g\) from \(S \times L\) into \(O\).

f) an initial state \(\sigma \in S\).

We write \(M = (I, O, S, f, g, \sigma)\).

[77]
fragile base class problem  Enhancements to a base class cause problems in the derived class. This is usually found when inheritance is used as a convenient tool to allow reuse.

function  Mathematics : a mapping between two sets of elements, domain and the range, such that each element of the domain corresponds to exactly one element on the range.
Programming : a procedure call which allows using the procedure name like a variable to assign a value to a program variable using an assignment operator.

functionality  A set of related software behaviour.


 hierarchy  Hierarchy refers to a relationship between entities, like parts of a system, where there are no loops in the relation. So if the relationship can be expressed as a predicate on pairs of the entities, or pairs of the parts, then $R(\alpha, \beta)$ allows the definition of levels by saying that
1. Level 0 is the set of pairs $\alpha$ such that there does not exist a $\beta$ such that $R(\alpha, \beta)$ and
2. Level $i$ is the set of pairs $\alpha$ such that
   a. there exists $\alpha \beta$ on level $i - 1$ such that $R(\alpha, \beta)$ and
   b. if $R(\alpha, \gamma)$ then $\gamma$ is on level $i - 1$ or lower. [124]

identifier  A symbol used to identify, or name, a unique item of data or part of a program. [56] (like an object, or class or method)
identity Definition 9.3 (Identity)

"Identity is the property of an object that distinguishes it from all other objects." [79]

IN The interpretation of physical phenomena into values in input registers is described by the relation IN.

Definition 9.4 (IN)

IN describes the relationship between the input devices and the monitored quantities

$IN \subseteq MS \times IS$

such that

$IN (m, i) \text{ is true}$

iff "i describes values of the input registers that are possible if m describes the values of the monitored quantities". [168]

For more details For details please refer to [135].

incident An undesired event which, although undesired, are expected and for which recovery is successful.

interface An interface defines a protocol of behaviour used by programs to communicate.

input value Input values are the representation of monitored quantities (see value).
Definition 9.5 (Input Value)
Quantities that can be read by computers in the system from input registers associated with the system.

"Let u be the number of input registers associated with the computers in the system S, and $F_i$ a selected set of time-functions. Then

1. inputs will be represented by a u-tuple, $i = (i_1, \ldots, i_u)$, where each $i_k$ is a member of $F_i$ and corresponds to the $k^{th}$ input register, and
2. the set of all i will be denoted by $I_S$.

[168]

invoke One program A can invoke another program B either through a call like call B(), through a function invocation like int v = B() where B is invoked without an explicit call or implicitly through an instanciation.

invocation stack Each time a method is invoked the callers data state at the time of invocation must be stored along with a pointer into callers code to where control must be returned. The invocation stack, also called the call stack, is a LIFO data structure that contains pointers to where this data is stored.

J

Java$^{TM}$ An Object Oriented programming language

Java$^{TM}$ interface An interface is a named collection of method definitions (without implementations). A Java$^{TM}$ interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all
of the methods defined in the interface, thereby conforming to certain behaviour.
[18]

Java™ package A package is a collection of related classes and Java™ interfaces[18].

see also package.

link Process of creating an executable module from object modules by “linking”
them together. The program which does this is commonly called a linker.

linker Program which creates an executable file from object code files.

load module Machine code program after being linked, ready to be loaded into a
region an executed

MACRO A series of program statements which can be treated as one statement.

maintenance history The history of changes which have been made to a piece of
software.

method Description of a classes behaviour in program code.

minimal subset The smallest subset of a system which still delivers a useful service
[121]

module A unit that can be constructed without knowledge of the internal struc-
ture(s) of other modules [125]

minimal increment The smallest addition to a functioning system which will result
in a still functioning software system [121]
monitored quantities Environmental quantities which a computer system is to measure [134]. For details please refer to Definition 9.6.

Definition 9.6 (Monitored Quantities)
Environmental values the system is intended to measure.

“Let $q$ be the number of quantities to be controlled in system $S$, and $F_M$ a selected set of time-functions. Then:

1. monitored quantities will be represented by a $q$-tuple called a monitored state function, $m = (m_1, \ldots, m_q)$, where each $m_i$ is a member of $F_M$ and corresponds to the $i^{th}$ monitored quantity; and

2. the set of monitored state functions will be denoted by $M_S$. ”

$^4[168]$

name See identifier

NAT The constraints placed on environmental quantities by nature, including existing systems, can be documented by means of a relation called NAT.

Definition 9.7 (NAT)
$NAT \subseteq M_S \times C_S$
such that
$NAT (m, c)$ is true
iff “the environmental constraints allow the controlled quantities to take on the
values described by c, if the values of the monitored quantities are described by \( m \). [168]

For more details For details please refer to [135].

**named reference** A program reference in the source code text like a variable or a method that has a name associated with it.

**Nearly Completely Decomposable** [4] Systems in which variable aggregations still yield satisfactory approximations when interactions between groups of variables are non null and arbitrary, but week compared with interactions within groups. [26].

**Network Computer** A general purpose end-use computing device that consists of:
a display device, a text input device, a system unit with processor, a network connection. NCs rely on their network for almost all their resources including permanent local storage [22].

**Object** For details please refer to Section 2.3 on page 17.

**Office Premium** The price charged for insurance[35] made up of:

a. pure risk premium  
b. an expenses loading  
c. a profit loading  
d. a contingency loading
**operational dependency** A dependency between the software and the data it operates on oracle. A mechanism which can determine if the output of a program is correct [174].

**oracle assumption** The belief that a tester is routinely able to determine whether the output of software under test is correct or not [174].

**ordinal** An ordinal scale of measure allows a ranking of objects according to some ordering criterion [15]. This is minimally necessary to be able to make a meaningful comparison between two systems.

**OUT** How output registers effect physical phenomena is described in the relation OUT.

**Definition 9.8  (OUT )**

OUT describes the relationship between the output devices and the controlled quantities.

OUT ⊆ O_S × C_S

such that

OUT (o, c) is true

iff “c describes values of the controlled quantities that are possible when o describes the values of the output quantities”. [168]

For more details For details please refer to [135].

**output values** Output values are the representation of controlled quantities (see value).
Definition 9.9 (Output values)

Let \( v \) be the number of output registers associated with the computers in the system \( S \), and \( F_O \) a selected set of time-functions. Then

1. outputs will be represented by a \( v \)-tuple, \( o = (o_1, \ldots, o_v) \), where each \( o_k \) is a member of \( F_O \) and corresponds to the \( k^{th} \) input register; and
2. the set of all \( o \) will be denoted by \( O_S \).

[168]

overloading  Allowing the same symbol to have two or more distinct meanings and allowing the meaning to be determined by the context [96].

package  A module.

perfective maintenance  Enhancement of the system to add new capabilities. These new capabilities include new user requirements and enhancements made in the interest of performance or efficiency.

polymorphism  The ability to take on several forms. In polymorphic languages methods can have the same names as long as they can be distinguished by their signatures.

preventative maintenance  A category of activities which change the software to improve future maintainability or reliability. Also included are activities which change the software to provide a better base for future enhancements[159].

problem domain  The part of the real-world the computer system must operate in.
production software Software which is operational and being used by a user community.

production environment Technical environment which is accessible by users in which production software resides while operational.

program Finite set of finite functions called instructions each of which has a finite domain [108]

\[
P = \{ z_i | \text{right}, i = 0, 1, 2,... \\
\text{where} \\
z \in I \\
I = \{ SetOfInstructions \} \]

program family “A set of one or more programs whose common properties are so extensive as to make it advantageous to study the common properties of the programs before analysing individual programs”[126];

programming plan Program fragment that represents stereotypic action sequences in programming[161][146]. In a simple form a programming plan could be a file reading loop while a more complex plan might be to instanciated a complex object.

quantity An amount that occurs in nature. A container filled with water is a quantity of water. (For details please refer to environmental quantity)

receiver parameter which determines which class to select for a method’s behaviour when classes are overloaded.
relation Mathematics: a relation R consists of the following:

1. a set A
2. a set B
3. an open sentence \( P(x,y) \) in which \( P(a,b) \) is either true or false for any ordered pair \( (a,b) \) belonging to \( A \times B \).

R is called a relation from A to B and is denoted by
\[ R = (A, B, P(x,y)) \]

reliability A statistical measure relating the system to the demands placed on it. A system is considered reliable if it is highly probable that when a demand is placed on the system, the system will perform satisfactorily. So software reliability is the probability of not encountering sequence of inputs which result in a failure. [125]

release control The process of grouping changes to production software into related bundles called releases which will be migrated into the production environment at the same time.

REQ The constraints placed on environmental quantities by the computer system define its permitted behaviour and can be documented by means of a relation called REQ, Definition 9.10.

**Definition 9.10** (**REQ**)

\[ REQ \subseteq M_S \times C_S \]

such that

\[ REQ(m,c) \text{ is true} \]

iff “the computer system should permit the controlled quantities to take on the values described by c when the values of the monitored quantities are described by m”. [168]
For more details *For details please refer to* [135].

**run-time** The period of time during which a program is executing in a computer.

**rules of programming discourse** Rules that specify the conventions in programming. These rules set up expectations in the minds of programmers about what should be in the program[161].

**sandwiching** A technique for reconciling a situation where two modules, A and B, can benefit from each other but cannot “use” each other for violation the uses hierarchy. One of the programs, B, is separated into B1, the part which benefits from using A and B2, the part A benefits from using. B1 is then moved higher and B2 lower in the hierarchy effectively sandwiching A between them.

**scope** A scope is a part of the program where the declaration of an identifier and what it is identifying is consistent. For example if a variable \( a \) is declared as `int` in method `ml_c` and as `char` in method `mC_m` the variable `a` has two scopes, one in `ml_c` and one in `mC_c`. These are two variables that have nothing on common other that the name and the name \( a \) can not be reused for any other purpose within these scopes, that is within `ml_m` or within `mC_m`.

**signature** A method’s signature is defined by the types of the parameters and the returned value. (Allowing for `void` as a returned type.)

**slice** Slicing is a filtering process where unwanted information is discarded. In program comprehension, slicing refers to a technique whereby only that part of a program that modifies a specified subset of data is visible. The visible part of the program is the slice that is relevant to the data being studied. In this
thesis the idea of a slice has been extended to include documentation. A slice
of documentation is the subset relevant to a maintenance task.

SOF The software requirements specify the behaviour needed to satisfy the system
requirements and the system design. The system requirements describes all the
restrictions placed on the system by the real world and by the system itself.
The system design describes how real-world values relate to values in machine
readable form. What remains is to describe how the machine readable input
variables relate to the machine readable output variables.

Definition 9.11 (SOF)

\[ \text{SOF} \subseteq I_S \times O_S \]

such that

\[ \text{SOF} (i, o) \text{ is true} \]

iff “the software could produce values described by o when the inputs are de-
scribed by i”. [168]

For more details For details please refer to [135].

software product All deliverables associated with an application including database
schema, configuration files, software and documentation.

software sponsor The body which fund the development and maintenance of a
software application.

software structure A partial description of a software system [122] showing it as a
collection of parts and showing some relations between the parts [124].

software system A set of related programs, modules and components.
shrink wrapped software Software that is sold to the general public like games or word processing software. This software often comes in a shrink wrapped package from which it derives its name.

state space The complete set of states that a program can pass through in all possible executions. Mathematically this is the range contained in the Cartesian product of the data space and the program [108].

state “The state of an object encompasses all of the properties of the object plus the current values of each of these properties.” [12]

static The named reference associates to a linked entity. In Java™ a static qualification indicates a class variable or method. Therefore, the named reference is associated with the class and allocated once per class at run-time, regardless of number of objects that are allocated.

structural design How a system is divided into parts, their interfaces and how the parts are related to one another.

system software Software associated with the operating system performing its required function

team A cohesive group of people working toward the same objective. More information about teams can be found in [33].

third party software Software purchased from a vendor other than the supplier of the computer hardware. Maintenance of this software is usually carried out by the vendor.

time-function A mathematical function which changes its value relative to time.
transparency  Refers to how much of the base machines behaviour is possible through the interface to a virtual machine.

trustworthiness  The probability that a serious error remains after the software passes a randomly chosen set of tests.

type  A type is determined by a representation of data and a set of operators that define it’s behaviour. The operators are defined it terms of a set of methods that operate on the representation.[137]. For the purposes of this thesis type and class are interchangeable.

undesired event  Events during the execution of the program which cause behaviour other than that desired. Two types : incidents and crashes [142]

uses  A relation between two programs where if program A uses B then A may rely on the correct execution of B. uses can also be formulated a “requires the presence of the correct version of”[121].

unanticipated UE  UEs for which there was no programmed response.

value  A representation of a quantity, a measurable quantity at every instant has a definite numerical value. A quantity of water, see quantity, can be represented using a scale of measure and a value that represents the quantity of water measures on that scale, litres say. Therefore a quantity of water can be represented as 1.35 litres of water.

variable  A storage location in a computer which is associated with a identifier and a type that can store values.
virtual machine An abstract programmable finite state machine

working document A document that is continuously updated to keep it current and continuously reviewed to keep them accurate. A simple example of this is the balance book for a bank chequeing account. Each time a cheque is written a record of the date, amount and the payee is kept and a new balance calculated. The balance is then periodically verified against bank records. This working document is current, accurate and verifiable. It is also kept current because it is useful to do so, and it is the easiest source of the information relevant to the audience.

work product Generic term for deliverables associated with a software product. Work products include documentation, code, data files, database specifications, test plans and any other deliverables.
Thesis Index

3270 Terminal Emulation, 82, 103, 113
A7E, 141, 188
ABEND, 38
abstract class, 30
abstract method, 31
abstract reasoning, 68
abstraction, 108, 108, 109, 111, 113
abstraction function, 153
acceptance, 3
acceptance testing, 89
access designator, 29
adaptive, 190
    maintenance, 43, 44, 46, 56, 65, 102,
    104
adaptive maintenance, 38, 60
API, 81, 103, 188
application data, 65
Application Programming Interface, 81
Application Window Toolkit, 103
as-needed, 62, 64, 69, 115, 171
assumptions, 80
AWT, 103, 113
    190, 209
batch, 63
beacon, 124, 129, 129
bottom-up, 64, 64, 65
bug, 35, 36, 133
business logic, 82, 82
business process, 44, 44
call, 198
call stack, 190, 198
canonical, 150
canonical trace, 150, 153
case
test, 91
catch, see Java\textsuperscript{TM} 121, 122
causal information, 68, 69
change request, 43, 45, 87, 89, 158, 159,
    190, 192
Change Request Log, 159
chunk, 59, 61, 61, 125, 130
class, 22, 22, 77, 149, 190, 190, 209
class variable, 23, 77
code text, 46-49, 60, 62, 70, 124, 191
coding standards, 125, 191
cognitive abilities, 68
cognitive load, 76
cohesion, 101, 115, 191
compile, 94, 191
completely decomposable, 119
completion, 150, 150
complexity, 13, 13, 16, 93, 119, 183
rule of increasing, 13
component, 191
components, 119
comprehension, 72, 76
as-needed, 69
just in time, 50
program, 51, 62, 65, 90, 124, 130
systematic, 87
top-down, 56
computer system, 136, 139, 154, 155, 191, 191, 194, 195, 200
concept assignment, 149
conjecture, 61
connection, 12, 80, 96, 101, 129
containment, 28, 77, 191
containment structure, 79
control, 116
control data, 66
control dependency, 81
control logic, 49, 49, 50
controlled quantities, 194, 202
controlled quantities, 191, 192
controlled quantity, 192
controlled state function, 192
corrected variable, 17
correct, 14, 35, 35, 37, 91, 192
corrective
maintenance, 43, 45, 46, 51, 56, 62,
64, 123, 131, 133, 159, 162
corrective maintenance, 35, 59
cost, 1
costs, 17
maintenance, 32, 165, 181
personnel, 16
coupling, 107, 115, 192
CR, 1, 43, 43, 45, 158–161, 163, 188
crash, 38, 38, 43, 46, 63, 154, 155, 157, 192, 209
data, 192
data base management system, 193
data declaration, 106
data definition, 106
data element, 106
data space, 88, 88, 94, 193, 193, 208
data state, 23, 88, 106
data structure, 106, 106

Jens Andexer

September 26, 2001
data type, 106
data value, 88, 193
data value, 195
DB2, 104
DBMS, 188
debug, 155
declaration, 26, 26, 193
deliverables, 193
delivery, 3
delocalised plans, 68
dependencies, 12, 13, 72
dependency, 69
control, 13, 72
data, 13
definitional, 13
functional, 13, 68
design, 15
rational, 136
Design Audit Trail, 156
design concept, 15
design error, 87
design errors, 87, 88
destruction, 22
destructor, 26
detect, 37
detection, 120
deterioration, 10, 15, 32, 33
development, 3
Dialog Management Systems, 103
Dialogue Management System, 194
discrepancy report, 194
display, 155
display method, 153
domain, 153, 196
knowledge, 55
maps, 52
plan, 52, 52
problem, 43, 43, 45, 52, 82, 93, 104, 105, 134, 194
user, 45, 55
domain knowledge
problem, 43
domain maps, 54
dynamic, 19, 23
dynamic process, 49, 50, 62, 72
effort, 16
maintenance, 9, 181, 185
engineering
software, 8
entropy, 134, 194
entry point, 194
environment, 194
environmental quantities, 194, 200, 205
equivalent, 150
Error, see Java\textsuperscript{TM} 93
error, 35, 35, 36, 41, 42, 46, 126, 194
error locator, 124
errors, 31
event, 194
Exception, see Java\textsuperscript{TM} 93
exception, 93, see Java\textsuperscript{TM} 93, see Java\textsuperscript{TM}
121
execution, 14, 22, 90, 91, 92, 153, 194,
208
executions, 91
exponential yo yo problem, 70
external entry point, 195
fail, 38
failure, 38
FIFO, 188
final state, 153
finally, see Java\textsuperscript{TM} 121
finite state machine, 195, 210
formal methods, 140, 195
fragile base class problem, 196
function, 196
functionality, 12, 15, 46, 52, 56, 62, 69,
77, 82, 196
garbage collection, 26
Graphical User Interface, 103
GUI, 82, 103, 105
handle, 37
handling, 120
hierarchical structure, 14, 14
hierarchy, 116, 196
uses, 118
history
maintenance, 7, 7, 14, 133
How conjectures, 61
hypotheses, 61, 61
hypothesis, 69
identifier, 193, 196, 209
identity, 196, 196
implementation, 76
IMS, 104, 140
IN, 147, 188, 188, 197, 197
In Code Documentation, 162
incident, 13, 37, 157, 197, 209
inheritance, 20, 27, 31, 77, 196
inheritance structure, 79
input value, 197, 197
inquiry episode, 61, 61, 68
inquiry episodes, 150
instance variable, 78
instance variables, 25
instanciated, 25
instanciation, 22, 22
instructions, 204
integration, 152
integration testing, 89, 152
interface, 80, see Java™ 80, 80, see
Java™ 114, 197
interface specification, 150
Internet, 174
invocation, 15
invocation stack, 94, 198
invoke, 198
IT, 188

java classes, 24
Java™, 198
catch, 121, 122
exception, 121
finally, 121
interface, 80, 114
JVM, 93
runtime exceptions, 93
throw, 93, 121
throwable, 121
try, 121, 122
Java™ interface, 198, 199
Java™ package, 199
Java™
  Error, 93
  Exception, 93
  exception, 93
  RunTimeException, 93, 93, 94
  Throwable, 93, 93
java.math, 81
JCL, 188
job stream, 63, 63
JRE, 188
JVM, see Java™ 93, 108, 188
knowledge
  domain, 55
  technical, 62–64
layout, 127
LD-relation, 153
LD-relations, 153
Lexicon, 154
LIFO, 189
link, 199
linker, 199, 199
load module, 199
LOC, 189
loss of transparency, 108
LXC, 12, 189
MACRO, 199
maintainability, 16, 95, 124
maintenance, 1, 3, 3, 7
  adaptive, 43, 44, 46, 56, 65, 102, 104
  corrective, 43, 45, 46, 51, 56, 62, 64,
  123, 131, 133, 159, 162
  costs, 32, 165, 181
  effort, 9, 181, 185
  history, 7, 7, 14, 133
  in-the-large, 7
  perfective, 43, 45, 51, 55, 156
  programmer, 59, 162
  programmers, 163
  software, 8, 10, 51, 63, 90, 96, 136,
  139, 149, 181, 183–186
  software under, 7
maintenance conversation, 11
maintenance history, 199
maintenance process
  quick fix, 42
  rational, 136
  software, 5, 41, 43, 90, 156, 158,
  181
maintenance statistics, 33
maps
domain, 52
message, 78
method, 25, 150, 199
minimal increment, 199
minimal subset, 199
module, 8, 14, 101, 101, 107, 199, 203
module guide, 150
Module Interface Specification, 149
Module Internal Design Document, 152
monitored quantities, 194, 197, 200, 200
monitored quantity, 200
monitored state function, 200
monitored variable, 17
multiple inheritance, 28
name, 200
named reference, 201
NAT, 145, 189, 189, 200, 200
natural, 135
NC, 189
Nearly Completely Decomposable, 201
nearly completely decomposable, 119
Network Computer, 201
object, 22, 195, 201
Object Oriented, vii, 2–5, 7, 18, 20–22,
  25–31, 55, 68, 69, 72, 73, 77–
  79, 82, 101, 103, 111, 112, 117,
office premium, 82, 201
on call, 155
on-line, 63
on-risk, 2
oracle, 202
oracle assumption, 91, 91, 202
ordinal, 202
OS platform, 80
OUT, 147, 189, 189, 202, 202
output quantities, 202
output values, 202, 202
overloaded, 27, 30, 204
overloading, 203
override, 30, 31, 78
package, 81, 111, 112, 149, 199, 203
perfective
  maintenance, 43, 45, 51, 55, 156
perfective maintenance, 38, 58, 62, 203
persist, 25
personnel
costs, 16
plan, 52, 52, 56, 61, 68, 68, 129, 130
domain, 52, 52
polymorphism, 26, 27, 203
presentation logic, 82
preventative maintenance, 203
principles of design for maintainability, 5
private, 74
problem
domain, 43, 43, 45, 52, 82, 93, 104, 105, 134, 194
domain knowledge, 43
problem domain, 55
production, 2, 3, 3, 7, 10, 89, 91, 94
production environment, 204
Production Failure Log, 156
Production Migration Log, 160
production software, 204
program, 150, 152, 193, 204, 208
program correctness, 88, 88
program domain, 59
program family, 204
programmer
  maintenance, 59, 162
programmers
  maintenance, 163
programming plan, 56, 204
programming plans, 58
programming standards, 125, 125
propagation, \textit{37}
public, 14, 74, 112
pure risk, 82
quantities
  controlled, 194, 202
quantity, \textit{204}, 209
quick fix
  maintenance process, \textit{42}
range, 153, 196
rating, \textit{55}
rational, \textit{135}, \textit{135}
  design, 136
  maintenance process, 136
readability, 127
receiver, \textit{204}
recording, \textit{124}
regression testing, \textit{89}
rehearsal, \textit{172}
relation, \textit{205}
  uses, 116, 117
release, \textit{9}
reliability, 88, \textit{88}, \textit{119}, \textit{205}
report, \textit{37}
reporting, 121
REQ, \textit{145}, 189, \textit{189}, 205, \textit{205}
requirements volatility, \textit{43}, \textit{44}, \textit{44}
restated, \textit{63}
reuse, 82, 115, 133, \textbf{133}, 196
review, \textbf{138}
role back, \textit{60}
Rule of increasing Entropy, 10
rules of programming discourse, \textbf{62}, 125, \textit{206}
run-time, 132, 133, \textit{206}
runtime, 94
runtime exceptions, \textit{see} Java\textsuperscript{TM} 93
RunTimeException, \textit{see} Java\textsuperscript{TM} 93, 94
sandwiching, \textit{206}
scope, 27, \textbf{27}, 77, \textit{206}
shrink wrapped software, 208
signature, 27, \textit{206}
single entry, \textit{126}
single exit, \textit{126}
single inheritance, \textit{28}
slice, 179, \textbf{179}, \textit{206}
SOF, 207, 207
software
  engineering, \textit{8}
  maintenance, 10, 51, 63, 90, 96, 136, 139, 149, 181, 183–186
  maintenance process, 5, \textbf{41}, 43, 90, 156, 158, 181

Jens Andexer

September 26, 2001
under maintenance, 8
software change process, 42
software engineering, 8
software maintenance, 8, 33, 35, 99, 132, 186
software maintenance process, 4, 99
Software Module Guide, 148
software module guide, 149
software product, 3, 8, 45, 136, 137, 142–144, 185, 207, 210
software sponsor, 207
software structure, 207
software system, 49, 52, 53, 55, 141, 149, 151, 207
specification, 88, 192
stack, 106
starting state, 153
state, 23, 23, 208
state space, 88, 88, 208
state value, 194
state value, 195
static, 19, 23, 25, 208
static text, 49, 50, 72
structural design, 208
structure, 15, 18
control, 125
directory, 70
documentation, 49
inheritance, 74
invocation, 14
program, 13, 14, 68, 78
software, 14, 15
uses, 14
sub-class, 27, 73
subclass, 74
super-class, 27, 73
System Design, 147
System Requirements Document, 144
system software, 208
systematic, 62, 62, 64, 69
team, 208
technical
knowledge, 62–64
technical knowledge, 64
terminating state, 153
test, 87, 92
case, 91
unit, 89, 92
test cases, 92
third party software, 208
throw, see Java™ 93, see Java™ 121
Throwable, see Java™ 93
throwable, see Java™ 121

time-function, 198, 203, 208

time-to-fix, 42

top-down, 55, 62, 62

trace assertion method, 150

traces, 150

transparency, 209

trap, 121

trustworthiness, 88, 88, 92, 95, 133, 209

trustworthy, 83

try, see Java™ 121, 122

type, 22, 28, 30, 79, 193, 209, 209

UE, 37, 37, 46, 52, 62, 64–66, 93, 121, 123, 124, 129, 155, 157, 158, 160, 189

UEs, 176

unanticipated UE, 120, 209

under maintenance

software, 8

Undesired event, 192

undesired event, 37, 120, 156, 189, 197, 209

unit

  test, 89, 92

unit test, 92, 152

unit test strategy, 152

unit testing, 89

user

  domain, 45, 55

user domain, 62

uses, 14, 116, 119, 151, 152, 183, 206, 209

  hierarchy, 118

  relation, 116, 117

structure, 14

uses hierarchy, 121

uses relation, 83

Uses-Relation Document, 150

value, 197, 202, 209, 209

variable, 13, 23, 106, 209

views, 179, 179


What conjectures, 61

white space, 127

Why conjectures, 61

work product, 138, 144, 210

working document, 158, 210

working documents, 138, 141

Y2K, 189
Y2K problem, 22, 80
Yo Yo Problem, 70
Bibliography


Jens Andexer

September 26, 2001


Jens Andexer

September 26, 2001


[61] P. Harrison, “Private Communication.” Mr. Harrison has more than 15 years experience as a software maintainer working on software with an excess of 1 million lines of code., 1998.


BIBLIOGRAPHY


Jens Andexer

September 26, 2001


---

Jens Andexer September 26, 2001


[170] M. Wang, “Program Functions and Abstraction Functions for the TTP Table
Holder,” Master’s thesis, Department of Electrical and Computer Engineering,
McMaster University, Hamilton Ont., December 1995. CRL Report No. 325
and 326.


[172] M. Weiser, “Program slicing,” IEEE Transactions on Software Engineering,

Architect at the IBM Center for Java Technology.


IEEE Transactions on Software Engineering, vol. 18, no. 12, pp. 1038–1044,


Jens Andexer

September 26, 2001


Appendix A

Software Risks

This section provides a small sample of the consequences of mistakes made during software maintenance. According to [119] the top 10 most expensive software errors were maintenance errors while [99] goes on to point out that

"the top 3 most expensive software errors involved a single line of source code and cost their organizations $1.6 billion, $900 million, and $245 million."

(Dollars are in USD)

Software maintenance require meticulous work. It would be easy to blame these errors on sloppy maintenance programmers, but their frequency implies most maintainers are sloppy programmers. Without software originally designed for maintainability along with effective supporting documentation, these types of errors will continue to be unacceptably frequent. The following are more examples for maintenance problems.

"Largest Computer Error in U.S. Banking History: US $763.9 billion."

"When Jeff Ferrera and Cindy Broadwater checked their checking balance at the First National Bank of Chicago, the automated voice gave it..."
as $924,844,208.32. More than 800 other folks had similar stories to tell. The sum total for all accounts was $763.9 billion. More than six times the total assets of the First Chicago NBD Corp. The Problem was attributed to a “computer glitch””  

David Kennedy

“The “glitch” was apparently the result of a programming change intended to support the new out-of-area ATM fees proposed by various banking groups. When the new transaction messages where introduced to the network, some systems took the strange new codes and transformed them into something they could understand: a posting of a huge credit to ones account.” Louis Koziarz in [115, Pg. 13]

“Computer Error Costs MCI $ Millions”

“The Washington Post, 29 March 1996, MCI reported that they will refund about $40 million due to a computer error. A billing error was uncovered by an investigative reporter from a local television station, WRIC in Richmond, VA. The reporters found that they where charged for 4 minutes after making a 2.5 minute call, leading to an in depth investigation.” Scott Lucero[115, Pg. 13]

“Another Banking System Hits the Dust”

“On November 30 1996, the Canadian Imperial Bank of Commerce Interact service was victimized by its attempted software upgrade, affecting about half of all would-be transactions across eastern Canada.”  

John C Bauer[116, Pg. 22]

“Fidelity Brokerage Computer Problems ”

“An article in the Wall Street Journal, 4 Nov 1996 describes a major problem for Fidelity Brokerage Services (a discount stock brokerage) in London. Very few details are given beyond “late bookings of dividends and other problems”, but it’s serious enough that more than 50 people
are working 14-hour days to sort through and correct three months of records manually. British authorities have forced FBS to stop taking new customers until the problems are solved.”  *George C. Kaplan*[116, Pg. 22]

“Computer Malfunction Causes Panic Selling on Hong Stock Exchange”

“A Computer glitch in the Hong Kong Stock Exchange caused panic selling on 12 December 1996 when its teletex information system incorrectly reported a drop of 515 points (or about 4%) of the Hang Seng Index during the opening minutes of trading.”  *Joel Chan*[116, Pg. 23]

“Computer Gives Thousands a Tax Break”

“It could be You! Hundreds of thousands of people are set to get a tax holiday after the Inland Revenue “lost” millions of tax records.

... The 5.2 million records from last financial year disappeared because the Inland Revenue’s aging mainframe computer could not link up properly to the new national Insurance computer PAYE contributions and salary records. The mainframe is therefore registering no tax or payment details for 1998-99.

... The system has hit such a crisis that the Inland Revenue plans to ask Dawn Primarolo, the paymaster general, for permission to close the files on the missing records, effectively writing off the missing tax.

An Inland Revenue spokesman said: “We haven’t lost teh records, we know they are in the system somewhere. We expect to find them and match them up eventually.””

[152]

“R/3-Upgrade is Expensive”
“A version change for the total solution R/3 of the software developer SAP requires an investment of between 1/4 to 1/3 of the original installation costs according to research gathered by AMR Research from 60 SAP application companies. Primary cost drivers are extensive tests and additional training. In addition, adapting the software during the original implementation make a release change an expensive undertaking.”

[109]
Appendix B

Published Estimates of Maintenance Costs

Table B.1: Published Estimates of Maintenance Costs

<table>
<thead>
<tr>
<th>Year</th>
<th>Source</th>
<th>Maintenance Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1969</td>
<td>[147]</td>
<td>Possibly as high as 40% to 60% for most companies who have had a computer systems effort for a number of years.</td>
</tr>
<tr>
<td>1971</td>
<td>[38]</td>
<td>About 50% of programming expenses.</td>
</tr>
<tr>
<td>1973</td>
<td>[68]</td>
<td>About 40% of programming resource.</td>
</tr>
<tr>
<td>1973</td>
<td>[1]</td>
<td>75%</td>
</tr>
<tr>
<td>1975</td>
<td>[107]</td>
<td>75% of DP personnel are occupied with maintenance</td>
</tr>
<tr>
<td>1975</td>
<td>[78]</td>
<td>Up to 40% of programming resources</td>
</tr>
<tr>
<td>1975</td>
<td>[52]</td>
<td>40% or more of the cost of development</td>
</tr>
<tr>
<td>1975</td>
<td>[14]</td>
<td>40% to 60% of software costs</td>
</tr>
<tr>
<td>1976</td>
<td>[42]</td>
<td>75% of costs</td>
</tr>
<tr>
<td>1976</td>
<td>[10]</td>
<td>Probably about 70% of overall costs</td>
</tr>
<tr>
<td>1976</td>
<td>[98]</td>
<td>In many organisations, at least 70% of time of analysts and programmers</td>
</tr>
<tr>
<td>1978</td>
<td>[94]</td>
<td>51% of time of systems and programming personnel</td>
</tr>
<tr>
<td>1978</td>
<td>[180]</td>
<td>67%</td>
</tr>
<tr>
<td>1979</td>
<td>[111]</td>
<td>66%</td>
</tr>
<tr>
<td>Year</td>
<td>Source</td>
<td>Estimate</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>1979</td>
<td>[47]</td>
<td>38%</td>
</tr>
<tr>
<td>1980</td>
<td>[93, pg. 153]</td>
<td>Often held at around 50% as a result of deliberately freezing enhancement work</td>
</tr>
<tr>
<td>1983</td>
<td>[58]</td>
<td>53% of software costs</td>
</tr>
<tr>
<td>1985</td>
<td>[10]</td>
<td>(Prediction) 60% of all (hardware and software) costs</td>
</tr>
<tr>
<td>1988</td>
<td>[110]</td>
<td>40% of software costs</td>
</tr>
<tr>
<td>1990</td>
<td>[117]</td>
<td>58% of software costs</td>
</tr>
<tr>
<td>1995</td>
<td>[178]</td>
<td>60% to 80%</td>
</tr>
<tr>
<td>1997</td>
<td>[159]</td>
<td>60% to 80%</td>
</tr>
</tbody>
</table>
Appendix C

Case Study: Ace Insurance
### 1 Documentation Standards

This section provides the standards to be used when producing documentation for a software product.

#### 1.1 Document Structure Recommendations

This section describes the relationships document parts have with each other. The structure of the documents is shown using BNF which is described in [114]. The documents described in this section are subdivided into progressively smaller parts. The smallest part of the document described in this section is a “simple documentation form” and can be recognised by the subscript “t”. These parts of the documentation appear only in the right hand column of the BNF description. Simple document forms are described in the section “Simple Documentation Forms” in the Standards for Document Content.

#### Structure of Documentation

<table>
<thead>
<tr>
<th>( \langle \text{softwareDoc} \rangle )</th>
<th>:= ( \langle \text{standardsDoc} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \langle \text{developmentDoc} \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \langle \text{maintenanceDoc} \rangle )</td>
</tr>
<tr>
<td>( \langle \text{standardsDoc} \rangle )</td>
<td>:= ( \langle \text{dctmn Standards} \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \langle \text{prgmg Standards} \rangle )</td>
</tr>
<tr>
<td>( \langle \text{dctmn Standards} \rangle )</td>
<td>:= ( \langle \text{dctmn Structure} \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \langle \text{dctmn Content} \rangle )</td>
</tr>
<tr>
<td>( \langle \text{prgmg Standards} \rangle )</td>
<td>:= ( \langle \text{cross Language} \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \langle \text{language Specific} \rangle )</td>
</tr>
<tr>
<td>( \langle \text{language Specific} \rangle )</td>
<td>:= ( \langle \text{individual Lang Std} \rangle )</td>
</tr>
</tbody>
</table>

---

Print Date: September 26, 2001

V02-R11 Home Owner Freedom System
Structure for System Requirements

\[\langle systemReqInfo \rangle ::= \langle uiDocumentation \rangle \]
\[\langle nonUiDocumentation \rangle \]
\[\langle uiDocumentation \rangle ::= \langle panelDoc \rangle \]
\[\langle VBBDoc \rangle \]

\[\langle panelDoc \rangle ::= \langle panel \rangle \]
\[\langle panel \rangle \langle panelDoc \rangle \]

\[\langle panel \rangle ::= \langle purposeHeading \rangle \]
\[\langle identifier \rangle \]
\[\langle language \rangle \]
\[\langle layout \rangle \]
Structure for System Design

\[
\langle \text{systemDesignInfo} \rangle ::= \langle \text{introduction} \rangle \\
\langle \text{inputVInfo} \rangle \\
\langle \text{outputVInfo} \rangle \\
\langle \text{softwareDesign} \rangle
\]
Structure for Module Guide

\[ \langle \text{moduleGuideInfo} \rangle ::= \langle \text{introduction} \rangle \\
\hspace{1em} \langle \text{moduleStructure} \rangle \\
\langle \text{moduleStructure} \rangle ::= \langle \text{behaviorHidingDoc} \rangle \\
\hspace{1em} \langle \text{interfaceHiding} \rangle \\
\hspace{2em} \langle \text{decisionHiding} \rangle \\
\langle \text{behaviorHidingDoc} \rangle ::= \langle \text{behaviorHidingDoc} \rangle \\
\hspace{1em} | \langle \text{behaviorHidingDoc} \rangle \\
\hspace{2em} \langle \text{behaviorHidingModule} \rangle \\
\langle \text{interfaceHidingDoc} \rangle ::= \langle \text{interfaceHidingDoc} \rangle \\
\hspace{1em} | \langle \text{interfaceHidingDoc} \rangle \\
\hspace{2em} \langle \text{interfaceHidingModule} \rangle \\
\langle \text{decisionHidingDoc} \rangle ::= \langle \text{decisionHidingDoc} \rangle \\
\hspace{1em} | \langle \text{decisionHidingDoc} \rangle \\
\hspace{2em} \langle \text{decisionHidingModule} \rangle \]

Structure for Module Interface Specification

\[ \langle \text{moduleInterfaceInfo} \rangle ::= \langle \text{introduction} \rangle \\
\hspace{1em} \langle \text{moduleInterfaces} \rangle \\
\langle \text{moduleInterfaces} \rangle ::= \langle \text{moduleInterfaces} \rangle \\
\hspace{1em} \langle \text{interfaceDoc} \rangle \\
\langle \text{interfaceDoc} \rangle ::= \langle \text{purposeHeading} \rangle \\
\hspace{1em} \langle \text{identifier} \rangle \\
\hspace{2em} \langle \text{syntax} \rangle \\
\hspace{2em} \langle \text{returnedVariables} \rangle \\
\hspace{2em} \langle \text{accessPrograms} \rangle \\
\hspace{2em} \langle \text{canonicalTraces} \rangle \]
Structure for Module Internal Design

\[
\langle \text{moduleInternalInfo} \rangle ::= \langle \text{introduction} \rangle \\
\langle \text{moduleInternals} \rangle
\]

\[
\langle \text{moduleInternals} \rangle ::= \langle \text{moduleInternals} \rangle \\
\langle \text{moduleInternalDoc} \rangle
\]

\[
\langle \text{moduleInternalDoc} \rangle ::= \langle \text{purposeHeading}_i \rangle \\
\langle \text{identifier}_i \rangle \\
\langle \text{invocationInfo}_i \rangle \\
\langle \text{parmList}_i \rangle \\
\langle \text{returned}_i \rangle \\
\langle \text{callSyntax}_i \rangle \\
\langle \text{mathSpecification}_i \rangle \\
\langle \text{internalDeclare}_i \rangle \\
\langle \text{diagram}_i \rangle
\]

Structure for Test Plan Documentation

not part of this case study

Structure for Design Audit Trail

\[
\langle \text{AuditTrailInfo} \rangle ::= \langle \text{developmentDecisions}_i \rangle \\
\langle \text{maintenanceDecisions}_i \rangle
\]

Structure for Lexicon
\[
\langle \text{lexiconInfo} \rangle \quad ::= \quad \langle \text{constantDoc} \rangle \\
\quad \quad \quad \quad \langle \text{functionDoc} \rangle \\
\quad \quad \quad \quad \langle \text{expressionDoc} \rangle \\
\langle \text{constantDoc} \rangle \quad ::= \quad \langle \text{constantDoc} \rangle \\
\quad \quad \quad | \quad \langle \text{constantDoc} \rangle \\
\quad \quad \quad \quad \langle \text{constantInfo} \rangle \\
\langle \text{constantInfo} \rangle \quad ::= \quad \langle \text{identifier} \rangle \\
\quad \quad \quad \quad \langle \text{description} \rangle \\
\quad \quad \quad \quad \langle \text{definition} \rangle \\
\langle \text{functionDoc} \rangle \quad ::= \quad \langle \text{functionDoc} \rangle \\
\quad \quad \quad | \quad \langle \text{functionDoc} \rangle \\
\quad \quad \quad \quad \langle \text{functionInfo} \rangle \\
\langle \text{functionInfo} \rangle \quad ::= \quad \langle \text{identifier} \rangle \\
\quad \quad \quad \quad \langle \text{description} \rangle \\
\quad \quad \quad \quad \langle \text{signature} \rangle \\
\quad \quad \quad \quad \langle \text{definition} \rangle \\
\langle \text{expressionDoc} \rangle \quad ::= \quad \langle \text{expressionDoc} \rangle \\
\quad \quad \quad | \quad \langle \text{expressionDoc} \rangle \\
\quad \quad \quad \quad \langle \text{expressionInfo} \rangle \\
\langle \text{expressionInfo} \rangle \quad ::= \quad \langle \text{identifier} \rangle \\
\quad \quad \quad \quad \langle \text{description} \rangle \\
\quad \quad \quad \quad \langle \text{definition} \rangle \\
\]

**Structure for Documented Entities**

\[
\langle \text{monitoredQInfo} \rangle \quad ::= \quad \langle \text{monitoredQDetails} \rangle \\
\quad \quad \quad | \quad \langle \text{monitoredQInfo} \rangle \\
\quad \quad \quad \quad \langle \text{monitoredQDetails} \rangle \\
\langle \text{monitoredQDetails} \rangle \quad ::= \quad \langle \text{purposeHeading} \rangle \\
\quad \quad \quad \quad \langle \text{identifier} \rangle \\
\]

Print Date: September 26, 2001    V02-R11    Home Owner Freedom System
\[
\begin{align*}
\langle & \text{description} \rangle \\
\langle & HwSw \rangle \\
\langle \text{controlledQInfo} \rangle & ::= \langle \text{controlledQDtls} \rangle \\
\langle \text{controlledQInfo} \rangle & \mid \langle \text{controlledQInfo} \rangle \\
\langle \text{controlledQDtls} \rangle & ::= \langle \text{purposeHeading} \rangle \\
\langle \text{identifier} \rangle \\
\langle \text{description} \rangle \\
\langle HwSw \rangle \\
\langle \text{inputVInfo} \rangle & ::= \langle \text{inputVDtls} \rangle \\
\langle \text{inputVInfo} \rangle & \mid \langle \text{inputVInfo} \rangle \\
\langle \text{inputVDtls} \rangle & ::= \langle \text{purposeHeading} \rangle \\
\langle \text{identifier} \rangle \\
\langle \text{description} \rangle \\
\langle \text{relationToTime} \rangle \\
\langle \text{dataRepresentation} \rangle \\
\langle \text{validValues} \rangle \\
\langle \text{outputVInfo} \rangle & ::= \langle \text{outputVDtls} \rangle \\
\langle \text{outputVInfo} \rangle & \mid \langle \text{outputVInfo} \rangle \\
\langle \text{outputVDtls} \rangle & ::= \langle \text{purposeHeading} \rangle \\
\langle \text{identifier} \rangle \\
\langle \text{description} \rangle \\
\langle \text{relationToTime} \rangle \\
\langle \text{dataRepresentation} \rangle \\
\langle \text{validValues} \rangle \\
\langle \text{modeClassInfo} \rangle & ::= \langle \text{modeClassInfo} \rangle \\
\langle \text{modeClassInfo} \rangle & \mid \langle \text{modeClassInfo} \rangle \\
\langle \text{modeClassDtls} \rangle & ::= \langle \text{purposeHeading} \rangle \\
\langle \text{identifier} \rangle \\
\end{align*}
\]
\[
\langle \text{description}_i \rangle \\
\langle \text{transition}_i \rangle \\
\langle \text{eventInfo}_i \rangle \\
\langle \text{modeInfo} \rangle ::= \langle \text{modeInfo} \rangle \\
| \langle \text{modeInfo} \rangle \\
\langle \text{modeDtls} \rangle \\
\langle \text{modeDtls} \rangle ::= \langle \text{identifier}_i \rangle \\
\langle \text{description}_i \rangle \\
\langle \text{values}_i \rangle \\
\langle \text{condition}_i \rangle \\
\langle \text{eventInfo} \rangle ::= \langle \text{eventInfo} \rangle \\
| \langle \text{eventInfo} \rangle \\
\langle \text{eventDtls} \rangle \\
\langle \text{eventDtls} \rangle ::= \langle \text{identifier}_i \rangle \\
\langle \text{fromEvent}_i \rangle \\
\langle \text{toEvent}_i \rangle \\
\langle \text{demandOprnInfo} \rangle ::= \langle \text{demandOprnInfo} \rangle \\
| \langle \text{demandOprnInfo} \rangle \\
\langle \text{demandOprnDtls} \rangle \\
\langle \text{demandOprnDtls} \rangle ::= \langle \text{identifier}_i \rangle \\
\langle \text{description}_i \rangle \\
\langle \text{modesAvailable}_i \rangle \\
\langle \text{requestDisplayDesc}_i \rangle \\
\langle \text{periodicOprnInfo} \rangle ::= \langle \text{periodicOprnInfo} \rangle \\
| \langle \text{periodicOprnInfo} \rangle \\
\langle \text{periodicOprnDtls} \rangle \\
\langle \text{periodicOprnDtls} \rangle ::= \langle \text{identifier}_i \rangle \\
\langle \text{description}_i \rangle \\
\langle \text{modesAvailable}_i \rangle \\
\langle \text{oprnStartStop}_i \rangle 
\]
2 Standards for Document Content
The following sections describe how some key aspects of this system are documented.

2.1 Simple Documentation Forms, Content
This section describes the contents of the basic building blocks used to describe the Home Owner Freedom system.

\(\text{tdf-1} \quad \langle\text{purposeHeading}_l\rangle\)
A descriptive heading briefly identifying the purpose of what is being documented.

\(\text{tdf-2} \quad \langle\text{identifier}_l\rangle\)
A character string name that identifies what is being documented.

\(\text{tdf-3} \quad \langle\text{description}_l\rangle\)
A description of what is being documented. This description can be natural language text but can and should, where possible, also use mathematics to provide a precise description.

\(\text{tdf-4} \quad \langle\text{coordinates}_l\rangle\)
Provides coordinate references for screen objects and the distance these are from a fixed point, usually the bottom left corner of the object being described.

\(\text{tdf-5} \quad \langle\text{layout}_l\rangle\)
The layout shows how the visual object will typically appear to the user. The layout also shows where the coordinate references appear.

\(\text{tdf-3} \quad \langle\text{crossreference}_l\rangle\)
The crossreference identifies where specific instances of VBBs or images appear on visual objects. The crossreference is in a tabular form. The columns contain the following information.
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Element</td>
<td>The Visual Element being referred to the instance name of the Visual Element</td>
</tr>
<tr>
<td>Instance</td>
<td>a reference to a screen coordinate that contains the value that must be assigned to $BLX_{eq}$ so the image or VBB will appear at the desired location within the visual object.</td>
</tr>
<tr>
<td>instance: BLX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a reference to a screen coordinate that contains the value that must be assigned to $BLY_{eq}$ so the image or VBB will appear at the desired location within the visual object.</td>
</tr>
<tr>
<td>instance: BLY</td>
<td></td>
</tr>
</tbody>
</table>

### 2.2 Documenting the User Interface

Screens are composed of two types of visual components. The simpler of these is a visual building block or VBB. A VBB describes the appearance of a portion of screen and its behaviour. A VBB can be used in another VBB or in the other screen composition component like a panel. Panels are collections of VBBs, text literals and images. Two factors differentiate between Panels and VBBs. First a panel cannot be part of another panel. Second, whereas any number of VBBs can appear in a Home Owner Freedom GUI window at any on time only one panel may do so.

#### 2.2.1 Simple Visual Building Block

A simple visual building block (VBB) does not contain any other VBBs. Below is an example of a simple VBB and a description of each part of it’s documentation.
One Character Text

**Identifier:** text1_vbb

**Description:**
Visual building block for a single character data input field.

**Image Layout:**

![Diagram of vbb-1](image_url)

**Co-ordinates:**

<table>
<thead>
<tr>
<th>X Coordinates Top</th>
<th>X Coordinates Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{TR} = x_{BL} + 28 \text{ pt.} )</td>
<td>( x_{BL} = .BLX_{eq} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y Coordinates Left</th>
<th>Y Coordinates Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_{BL} = .BLY_{eq} )</td>
<td>( y_{TR} = y_{BL} + 14 \text{ pt.} )</td>
</tr>
</tbody>
</table>

**Cross Reference:**
none

### Legend for the Simple VBB

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X coordinate ( x_{TR} ).</td>
</tr>
<tr>
<td>B</td>
<td>Construction line for X coordinate ( x_{TR} ).</td>
</tr>
<tr>
<td>C</td>
<td>Exterior boundary of the VBB’s screen footprint.</td>
</tr>
</tbody>
</table>

vbb-1 etc. entity references for the documented part of the VBB.

### 2.2.2 Composite VBB

A composite VBB is made up Images from other VBBs and text. See panels for a description of their construction.
2.2.3 Panel

HOF Risk Data Input

- **Identifier:** G1HF325E
- **Language:** English

**Coordinates:**
see Simple Documentation Forms for content

**Crossreference:**
see Simple Documentation Forms for content
**Legend for Panel**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Axis</td>
</tr>
<tr>
<td>B</td>
<td>numeric identifying the co-ordinated. These are unique along each side of the layout, making $x_i^k$ unique for the Layout.</td>
</tr>
<tr>
<td>C</td>
<td>T : top, B : bottom, L : left, R : right.</td>
</tr>
<tr>
<td>D</td>
<td>Exterior boundary of the panel’s screen footprint.</td>
</tr>
<tr>
<td>E</td>
<td>Text literals.</td>
</tr>
<tr>
<td>F</td>
<td>The <em>Acelcon</em> Image.</td>
</tr>
<tr>
<td>G</td>
<td>Construction lines. These do not appear on screen but help align screen parts in the layout.</td>
</tr>
<tr>
<td>H</td>
<td>Exterior boundary of the riskAddress, VBB’s screen footprint.</td>
</tr>
<tr>
<td>I</td>
<td>The postCode, VBB depends on its format on the risk country data. Since an incorrect image might lead to confusion a space the size of the VBB’s maximum footprint size is reserved on screen.</td>
</tr>
<tr>
<td>J</td>
<td>Text literal Screen identifier.</td>
</tr>
</tbody>
</table>
2.3 Standards for Document Appearance

Standards for the physical appearance of documents are described in this section.

2.3.1 Notation Standards

This section describes the notation and references that will be used throughout the documentation of the Home Owner Freedom system.

2.3.1.1 Abbreviations

**Document Abbreviation and Names**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Document Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Design Audit Trail</td>
</tr>
<tr>
<td>CL</td>
<td>Change Request Log</td>
</tr>
<tr>
<td>CR</td>
<td>Change Request</td>
</tr>
<tr>
<td>FL</td>
<td>Production Failure Log</td>
</tr>
<tr>
<td>ID</td>
<td>Module Internal Design</td>
</tr>
<tr>
<td>LX</td>
<td>Lexicon</td>
</tr>
<tr>
<td>MG</td>
<td>Module Guide</td>
</tr>
<tr>
<td>MI</td>
<td>Module Interface Specification</td>
</tr>
<tr>
<td>ML</td>
<td>Production Migration Log</td>
</tr>
<tr>
<td>SR</td>
<td>System Requirements</td>
</tr>
<tr>
<td>ST</td>
<td>Standards Manual</td>
</tr>
<tr>
<td>TP</td>
<td>Test Plan Documentation</td>
</tr>
<tr>
<td>UR</td>
<td>Uses Relation Document</td>
</tr>
</tbody>
</table>

**Entities Used to Describe Software Parts**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Documented Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bp</td>
<td>Blue Print</td>
</tr>
</tbody>
</table>

Print Date: September 26, 2001       V02-R11       Home Owner Freedom System
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ck</td>
<td>Constant</td>
</tr>
<tr>
<td>cq</td>
<td>Controlled environmental Quantity</td>
</tr>
<tr>
<td>cr</td>
<td>Change Request</td>
</tr>
<tr>
<td>df</td>
<td>Demand FuNction</td>
</tr>
<tr>
<td>er</td>
<td>ERRor token</td>
</tr>
<tr>
<td>ev</td>
<td>EVent</td>
</tr>
<tr>
<td>ex</td>
<td>EXpression</td>
</tr>
<tr>
<td>fn</td>
<td>FunctioN</td>
</tr>
<tr>
<td>im</td>
<td>IMage</td>
</tr>
<tr>
<td>in</td>
<td>module INput</td>
</tr>
<tr>
<td>ic</td>
<td>InterFace Class</td>
</tr>
<tr>
<td>ip</td>
<td>InterFace Package</td>
</tr>
<tr>
<td>iv</td>
<td>Input Variable</td>
</tr>
<tr>
<td>md</td>
<td>Mode</td>
</tr>
<tr>
<td>mc</td>
<td>Mode Class</td>
</tr>
<tr>
<td>ml</td>
<td>MoDuLe</td>
</tr>
<tr>
<td>mq</td>
<td>Monitored environmental Quantity</td>
</tr>
<tr>
<td>nfc</td>
<td>Non Funtional Consieration</td>
</tr>
<tr>
<td>ot</td>
<td>Module Output</td>
</tr>
<tr>
<td>ov</td>
<td>Output Variable</td>
</tr>
<tr>
<td>pL</td>
<td>Language specific Programming standard</td>
</tr>
<tr>
<td>pX</td>
<td>cross language Programming standard</td>
</tr>
<tr>
<td>pf</td>
<td>Periodic Function</td>
</tr>
<tr>
<td>pfL</td>
<td>Production Failure Log entry</td>
</tr>
<tr>
<td>pl</td>
<td>Panel</td>
</tr>
<tr>
<td>pml</td>
<td>Production Migretion Log enty</td>
</tr>
<tr>
<td>pv</td>
<td>Parameter Variable</td>
</tr>
<tr>
<td>sdc</td>
<td>Software Design Consideration</td>
</tr>
<tr>
<td>tdf</td>
<td>Simple Documentation Form</td>
</tr>
<tr>
<td>tp</td>
<td>Type</td>
</tr>
<tr>
<td>uec</td>
<td>Undesired Event Incident</td>
</tr>
<tr>
<td>uei</td>
<td>Undesired Event Crash</td>
</tr>
<tr>
<td>use</td>
<td>uses relation</td>
</tr>
<tr>
<td>vbb</td>
<td>Visual Building Block</td>
</tr>
</tbody>
</table>

**Entities Used to Describe Documentation Parts**
### 2.3.2 Cross References

This section describes how parts of the system being documented can be referenced.

**Entity reference**

**Description**

Each entity reference uniquely identifies a part of the documentation for all time. Therefore, once used an entity reference cannot be reused even if the entity it is associated with has been removed from the system. This is so that old and new versions of the documentation can be used without the confusion resulting from ambiguous references. The entity reference adheres to the following format:

**Format:**

```
aaa-1...n-e
```

**Legend:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
<td>identifies the entity being documented, see dns-2 above for these.</td>
</tr>
<tr>
<td>1...n</td>
<td>A counter for each entity (for entities see dns-2). This counter is unique, if an entity is dropped from the system the counter can not be re-used.</td>
</tr>
<tr>
<td>e</td>
<td>number identifying the aspect of the entity being documented</td>
</tr>
</tbody>
</table>

### 2.3.3 Bracketing and Subscripts

\( a_{in} \quad a_{in} \) indicates that \( a_{n} \) is an input variable to the system.
\( b_{ov} \)  
\( b_{ov} \) indicates that \( b_{ov} \) is an output variable from the system.

\( m_{mq} \)  
\( m_{mq} \) is a monitored variable.

\( c_{cq} \)  
\( c_{cq} \) is a controlled variable.

\$yes\$  
Some non-numeric data items have been given mnemonic names. The values these names can take are defined, the definition can be referenced through the lexicon.

\( K_{ck} \)  
Denotes a constant. Complex terms used in more than one place in the document are defined in the Lexicon.

\( F_{fn}(x) \)  
Denotes a function. Complex terms used in more than one place in the document are defined in the Lexicon.

\( T_{tp} \)  
Denotes a type. Complex terms used in more than one place in the document are defined in the Lexicon.

\(*\*x\)  
System states are denoted by \(*\*state\).

\( x_{\text{mds}} \)  
System States have been grouped into modes that are denoted by \( x_{\text{mds}} \). A mode is a class of system states and the current mode is defined by the history of events that occurred in the program.[63]

\( x_{**mc**} \)  
System modes have been grouped into mode classes that are denoted by \( x_{**mc**} \).
[j]  j is a subscript to an array. For example A[j] means the jth element in array A.
Rages are separated by a “;” (colon) such that A[5:9] means array elements 5 to 9.
Individual array elements can be grouped by separating them with a “,” (comma) such that A[5,9] means elements 5 and 9.

⟨j⟩  j is a subscript to a string. For example S⟨j⟩ means the jth character in string S.
Rages are separated by a “;” (colon) such that S⟨5:9⟩ means the sub-string from position 5 to 9 while S⟨j for 9⟩ means the sub-string beginning with character j and containing characters to j+9-1.

∃i  The symbol representing the universal quantifier “there exists”.
I.e. ∃i, B[i]=x Is the value x to be found in B ¹

∀i  The symbol representing the universal quantifier “for all”.
I.e. ∀i:0 < i ≤ n, B[i] = x The value x is in B.

∈  The symbol representing the relation “in”.
I.e. “1” ∈ {2,4,1,5,7} 1 is in the set of numbers

@T_{t_0}(x)  Means event x becomes True

@F_{t_0}(x)  Means event x becomes False

l := r  r is assigned to l

l := r  l is assigned to r

'e  before event e,

¹From [129].
after event e
I.e. \( @F_{\text{mq}}('\text{button1}_{\text{mq}} = \text{button1}_{\text{mq}}') \)
means; if the state of \( \text{button1}_{\text{mq}} \) before it was measured
is not the same as the state of \( \text{button1}_{\text{mq}} \) after it was
measured then the button was pushed. Notice, if the
button is pushed and held only one event is registered.
But another event is registered when the button is re-
leased.

\( \neg(a) \)

negation or NOT
I.e. \( \neg(x = y) \) is true when value x is not equal to y.

\( > n, t \leq_{in} \)
n is the name of an input variable to a module and t is
its type.
The value passed in this argument will be used but will
not be changed by the module.

\( > n, t \leq_{ot} \)
n is the name of an output variable to a module and t
is its type.
The value passed in this argument may be used but will
also be changed by the module.

\( x_{\text{er}} \)
Error token x. This is a token that identifies a class
or errors that can be found in the lexicon’s UE section.
If the detected error is handled by the Home Owner
Freedom system then the error will be in the incidents
section, otherwise it may be in the crash section.

\( i_{\text{uei}} \)
Undesired Event, i, that constitutes an incident. These
are documented in the Lexicon.

\( c_{\text{uec}} \)
Undesired Event, c, that constitutes a crash. These are
documented in the Lexicon.
\( a_{\text{hw}} \)  

\( a \) is a Hardware reference. Often aspects of monitored and controlled environmental quantities are best described together, like in a screen layout. This reference provides a connection between an aspect of the quantities and a common documentation of this aspect.

\( : \)  

A separator used in writing names of reusable parts. The left side of this separator identifies the instance of reusable part of the documentation. To the right of the separator is the identifier of the part of the reusable entity being referred to. For example

\[
\text{keyedText}_{mq}[1] = t2:\text{inputText}_{mq}
\]

means that the first element of \( \text{inputText}_{mq} \) takes on the value of \( \text{keyedText}_{mq} \) from the \( t2 \) instance of the text1 VBB.

\( \langle a \rangle \)  

A metalinguistic variable whose values are sequences of symbols [114].

\( ::= \)  

A metalinguistic connective used to show how a the metalinguistic character on the left of the connective symbol is constructed in BNF notation[114].

---

### 2.3.4 Page Layout Standards

The following standards describe how document pages should appear.

#### Page Header Information

**Description**

Page headers provide information about the context within which entities are documented on a page. For example in figure below the header provides the following information:

**Example:**

<table>
<thead>
<tr>
<th>Visible Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{text1}_{vbb}</td>
</tr>
</tbody>
</table>

---

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
Legend:

<table>
<thead>
<tr>
<th>Location</th>
<th>Header Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>SR</td>
<td>document abbreviation</td>
</tr>
<tr>
<td></td>
<td>page number</td>
<td>running page number that starts at one for each document work product.</td>
</tr>
<tr>
<td>right, first line</td>
<td>Visible Characteristics</td>
<td>The section in the current document.</td>
</tr>
<tr>
<td>second line</td>
<td>text1&lt;/sub&gt;</td>
<td>The VBB being documented.</td>
</tr>
</tbody>
</table>

Page Footer Information

Description

Page footers provide data about how current the information contained in the documentation is:

Example:

| Print Date: September 26, 2001 | V02-R11 | Home Owner Freedom System |

Legend:

<table>
<thead>
<tr>
<th>Location</th>
<th>Header Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Print Date</td>
<td>Date this page was printed</td>
</tr>
<tr>
<td>center</td>
<td>Vmn</td>
<td>V: version, followed by the version of the software product</td>
</tr>
<tr>
<td></td>
<td>Rmn</td>
<td>R: release, the release of the software product. If no release concept is being used this can be replaced by a date.</td>
</tr>
<tr>
<td>right</td>
<td>Home Owner Freedom</td>
<td>system being documented</td>
</tr>
</tbody>
</table>

2.3.5 Document Formats

Two types of headings are used in the documentation. The first type of heading is the document heading, the second type of heading is the entity group heading.
Document Headings

Description
This type of heading identifies a group of related document parts, like “Standards for Language Usage” which is then subdivided into two more detailed groups “Documentation Language” and “Frequently used Words”. Recognisable by a strut running the width of the text this heading’s section number may change when new sections are added to the document.

Example:

3 Standards for Language Usage

Entity Related Headings

Description
An entity related heading is used to identify parts of the document that are related and have the following form. In this type of heading the entity reference does not change when sections are added.

Format:

<table>
<thead>
<tr>
<th>er</th>
<th>t...t</th>
</tr>
</thead>
</table>

Legend:

<table>
<thead>
<tr>
<th>er</th>
<th>entity reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>heading text</td>
</tr>
</tbody>
</table>

Example:

In this example the text1 VVB’s is given a descriptive name.

One Character Text

Print Date: September 26, 2001 V02-R11 Home Owner Freedom System
3.1 Simple Documentation Forms

The documentation of software entities is done with the following two forms.

das-5

das-5-1 **Description**

Often all the information necessary for a description can fit onto one line.

das-5-2 **Format:**

<table>
<thead>
<tr>
<th>er</th>
<th>i...i:</th>
<th>t...t</th>
</tr>
</thead>
</table>

das-5-3 **Legend:**

where:
er entity reference
i information identifier
t information text

das-5-4 **Example:**
In this example the text1 VBB is named with an alpha-numeric identifier.

<table>
<thead>
<tr>
<th>vbb-1.1</th>
<th>Identifier:</th>
<th>text1</th>
</tr>
</thead>
</table>

das-5

das-5-1 **Description**

When a description cannot fit onto one line the following form is used.

das-5-2 **Format:**

<table>
<thead>
<tr>
<th>er</th>
<th>i...i:</th>
</tr>
</thead>
</table>
| tttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
where : 
er    entity reference  
i     information identifier  
t     information text

das-5-4 Example:
In this example the text1 VBB is given a description.

<table>
<thead>
<tr>
<th>vbb-1.2</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Visual building block for a single character data input field.</td>
</tr>
</tbody>
</table>
4 Standards for Language Usage
This section provides guidelines for the language documentation is to be written in and for how that language is to be used.

4.1 Documentation Language
The original version of all parts to the Home Owner Freedom software product will be written and maintained in English. This includes, all programming comments, names or other references in the body of the code text.
Documents may be duplicated or translated to other languages but these translations must be considered copies of the original.

4.2 Frequently used Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>JAR</td>
</tr>
<tr>
<td></td>
<td>Java ARchive</td>
</tr>
<tr>
<td>N</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>Network Computer</td>
</tr>
<tr>
<td>P</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>Personal Computer</td>
</tr>
<tr>
<td>S</td>
<td>SNA</td>
</tr>
<tr>
<td></td>
<td>Simple Network Architecture</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
dws-1  TCP/IP
Transmission Control Protocol / Internet Protocol

VPN
Virtual Private Network

WWW
World Wide Web

4.3 Frequently used Words

Associate User
A user who is an employee of one of A.E Insurance’s franchise agencies.

Class
Definition of the data and methods an object that instantiates that class will have.

Corporate User
A user of a computer asset who is an employee of A.E insurance.

End Point
The point in the code where control is returned the invoking program, be it application code, the operating system or whatever. Can be found in any program by searching for “>> END”.

Entity
A identifiable part to which information can be associated. There are two types of entities used in this documentation: Entities Used to Describe Software Parts, see dns-1 for a complete list.
Entities Used to Describe Documentation Parts, see dns-2 for a complete list.

**I**

dws-3  
**internet**  
A colloquial term used to describe a very large group of interconnected networks also referred to as the World Wide Web.

dws-15  
**intranet**  
A TCP/IP network that is entirely under the control of one organisation, like $A_cE$.

**J**

dws-14  
**Java Archive**  
A compression mechanism for Java$^{TM}$ class files.

**M**

dws-14  
**Method**  
Description of a classes behaviour in program code.

dws-5  
**Module**  
A work assignment for an individual or Team.

**N**

dws-17  
**Network Computer**  
A Network Computer (NC) is a general end-user device designed with the characteristic that it gets nearly all its resources from other systems using a network. Because there is no local permanent storage, such as a hard disk, these resources including its operating system, applications and application data.

**P**

dws-4  
**Package**  
A package is a set of closely related classes. See also Module.

dws-16  
**Personal Computer**  
A PC general end-user device that can operate independently from the network. The PC has its physical resources local, including permanent storage. A PC, therefore, has its operating system stored locally and can have applications and data also stored locally.
dws-7

Restart
The ability for a program to be restarted at a point part way through its execution. Some long running programs must have the ability to be stopped, i.e. in the case of a UE, and restarted at the point where execution where interrupted without the loss of data or what state the execution was in. Programs that have this ability are called restorable.

S

Session
The time a user is logged on to the Home Owner Freedom system.

Dws-6

Start Point
The code line where the code that describes the programs primary behaviour begins. Can be found in any program by searching for “>> START”.

Dws-11

Individual User
Individual who gains access to A.E systems through the internet.

V

Virtual Private Network
A VPN is an extension of the intranet using the internet. A secure connection is built up between the A.E intranet and an associate, trusted, user to allow them access to Home Owner Freedom.

W

World Wide Web
See Internet
5 Programming Standards
This section provides the standards to be used when producing program text for a software product.

pX-1 Cross Programming Language Standards
As some software parts must be implemented in more than one technology their naming must be standardised across the technological platforms.

pX-1-1 Naming:

pX-1-1.1 Panel Naming
Panel names must be 8 characters long to allow for mainframe technologies. Panels definitions in IMS/DC or CICS, for example, may only have 8 character long names. The following convention will be used:

pX-1-1.2 Format:

<table>
<thead>
<tr>
<th>char</th>
<th>Content</th>
<th>Possible Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U.I.</td>
<td>E</td>
<td>3270 Screen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G</td>
<td>GUI Screen</td>
</tr>
<tr>
<td>2</td>
<td>U.I.</td>
<td>1</td>
<td>AWP</td>
</tr>
<tr>
<td></td>
<td>Technology</td>
<td>2</td>
<td>Bit Blast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>IMS/DC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>CICS</td>
</tr>
<tr>
<td>3,4</td>
<td>System Id.</td>
<td>CI</td>
<td>Customer information System</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HF</td>
<td>Home Owners Freedom</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GF</td>
<td>Home Owners General</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CO</td>
<td>Common to all Systems</td>
</tr>
<tr>
<td>5-7</td>
<td>Panel Id.</td>
<td>0-999</td>
<td>unique running alphabetic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-Z</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>User</td>
<td>e</td>
<td>English</td>
</tr>
<tr>
<td></td>
<td>Language</td>
<td>f</td>
<td>German</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td>Language Independent</td>
</tr>
</tbody>
</table>
Where Content is:

- U.I Environment Technical user interface environment.
- System Id. Identifies systems uniquely.
- Panel Id. Identifies all panels in a system uniquely.
- User Language User Language the screen is coded in.

pX-1-2 Documenting Code Changes

Code changes can only be made by setting the original code into comment and adding the new code. The programmer should make an effort to comment the changes so that the documentation of the changes does not interfere with the overall appearance of the code. Where space allows the insertion or deletion of code and its commenting can be done on the same line. Code that is set into comment must have each code line commented out separately, no block comments that run over more than one line.

The following legend is used in the coding examples below.

/* : start of comment in any language.
*/ : end of comment in any language.
cc : century portion of change date.
yy : year portion of change date.
mm : month portion of change date.
dd : day portion of change date.
p^n : the programmers user id. for identification purposes.
r^n : change request initiating the change.
d^n : a description of the impact the change request had on the program and dependencies the changes have with other parts of the system.

pL-1-2.1 Single line code change

Single line code changes can be identified at the end of the change line if there is space, otherwise the change should be treated as a block change.
Example:

```c
if (a==b) /* insert: ccymmd rrrrr ppppp */
/* if (a==c) delete: ccymmd rrrrr ppppp */
```

Start of block code change
The start of a code change must be identified by the following comment line:

Example:

```c
/* Start insert: ccymmd rrrrr ppppp */
/* Start delete: ccymmd rrrrr ppppp */
```

End of block code changed
The end of a code change must be identified by the following comment line:

Example:

```c
/* End insert: ccymmd rrrrr ppppp */
/* End delete: ccymmd rrrrr ppppp */
```

Entries to the Change Log
Each method or sub-program must have a change log that describes all changes made to the program in chronological order. This log must be in the following format and information:

Example:

```c
/* date cr pgmr Description */
/* ccymmd rrrrr ppppp dddddddd ... dddddddd */
/* dddddddd ... dddddddd */
/* ... */
/* dddddddd ... dddddddd */
```
pX-1-3  **Error Reporting**
Errors must be reported to output devices with the following information:

- **UE class code**: The code identifying the class of the UE. See the lexicon for these.
- **UE message**: The message text associated with the UE class.
- **UE Detection Reference**: A unique reference that identifies in the code text and in the program specification where each individual UE is detected. Each UE detected in the system has one of these references and no two references are the same even polymorphic methods do not share the same references.

pL-1  **Language Specific Standards**
The programming standards in this section apply to specific programming languages.

pL-1-1  **Java™ Programming Standards**
The following standards should be adhered to when writing new code or modifying existing Java™ program code.

pL-1-1-1  **Naming:**

pL-2-1.1  **Variables**
Letters in the first word must be lower case, second and following words have the first letter capitalised. The letters: “i”, “I”, “o” may not be used as variable names.

pL-1-1.2  **Methods**
First letter must be lower case an first letter of each following word must be capitalised.

pL-1-1.3  **Objects**
Letters in the first word must be lower case, second and following words have the first letter capitalised.

pL-1-1.4  **Classes**
All words in the name, including the first word must have the first letter upper case and the others lower case.
pL-1-1-1.5 Packages
Packages names are made up of words separated by periods, all lower case.

pL-1-1-2 Control Structures
All control structures must begin with an open brace and end with a closed brace. All control structure should adhere to the guideline of single and entry single exit. This means there should be exactly one point where control enters the control structure and one point where control leaves the control structure. The entry point should be at the top of the control structure and the exit point should be at the bottom. Exceptions to this rule must be clearly documented in the code.

pL-1-1-2 Methods
pL-1-1-2.1 Opening brace
On same line as loop statement.

pL-1-1-2.2 Statements
Indented zero spaces from the margin.

pL-1-1-2.3 Closing brace
On line following last executable line in same column as the start of the opening statement.

pL-1-1-2.4 Start Point
The code line where the code that specifies the programs primary behaviour will be called the program’s start point. The start point of the program must be clearly identified with the reference ‘’’START’’’ set in a comment. If there is more than one start point the start points must be uniquely numbered with the reference ‘’’STARTnn’’’ where nn is a number that makes the reference unique. Searching the program for ‘’’START’’’ should find all start points.

pL-1-1-2.5 End Point
The point in the code where control is returned the invoking program, be it application code, the operating system or whatever, will be called the program’s end point. The end point of the program must be clearly identified with the reference “>> END <<” set in a comment. If there is more than one end point the end points must be uniquely numbered with the reference “>> ENDnn <<” where nn is a number that makes the reference unique. If the end point is coupled with a specific start point then this coupling should be documented in the comment. Searching the program for “>> END” should find all start points.

**Loops**

**Opening brace**
On same line as loop statement.

**Statements**
Indented three spaces.

**Closing brace**
On the line following last executable line in same column as the start of the opening statement.

**Long Loops**
If the loop contains more that 20 executable statements the end brace must also have a comment that identifies the loop it terminates.

**Nested Loops**
If there is a loop within a loop this will be referred to as a nested loop (see also If conditions) with 2 nesting levels, level 1 is the outer loop and 2 is the inner loop. If loops are nested the closing brace must be commented to identify which loop it terminates.

**If condition**

**Opening brace**
On same line as condition.

**statements**
Indented three spaces.

**Closing brace**
On the line following last executable line in same column as the start of the opening “if”.

**Long blocks**
If the block of code for which the condition is the guarding condition is more that 20 statements long then the terminating brace must be commented to identifying the condition when there is no else condition.

pl-1-1-4.5 **else**
All if statements must have an else, even if null.

pl-1-1-5 **Else**

pl-1-1-5.1 **Commenting**
The “if” to which the “else” belongs must be identified in a comment immediately above the “if” when there are more than two nesting levels.

pl-1-1-5.2 **Opening brace**
On same line as “else”.

pl-1-1-5.3 **statements**
Indented three spaces.

pl-1-1-5.4 **Closing brace**
On line following last executable line in same column as the start of the opening “else”.

pl-1-1-5.5 **Long blocks**
If the block of code for which the “else” is the guarding condition is more that 20 statements long then the terminating brace must be commented to identifying the “else” to which it belongs.

pl-1-1-6 **Switch block**

pl-1-1-6.1 **switch**
On its own line.

pl-1-1-6.2 **Opening brace**
On same line as switch.

pl-1-1-6.3 **case**
Indented three spaces.

pl-1-1-6.4 **statements**
Indented three spaces beyond case.

pl-1-1-6.5 **Closing brace**
On line following last executable line in same column as the start of the opening “switch”.

pl-1-1-6.6 **default**
All switch blocks must have a default, even if it is null.
Program Example

The following is a sample Java™ program that can be used as a guide
to the appearance of a program as text.

Example:

```java
package standards;

/* - Class: Foo */
/* - Purpose: example */
/* - change Log: */
/* - date cr pgmr Description */
/* - 19991117 cr1348 rd123 logic change to take previous */
/* - 19991117 CR1350 rd182 element change to that found */
/* - 19991117 CR1350 rd182 brought code inline with new naming */
/* - standard */

public class Foo {

    /* - Method: newMethod */
    /* - Purpose : example */
    /* - */

    public void newMethod (int month, int day) {
        int j;
        int z=0;
        int intArrayMax = 100
        intArray = new int[intArrayMax];
        int intArrrayCols = 5; /* 19991117 CR1350 rd182 */
        int intArrrayRows = 25; /* 19991117 CR1350 rd182 */
        int lv_colNotEmptyAt; /* 19991117 cr1348 rd123 */
        int lv_colNotEmptyNext; /* 19991117 cr1348 rd123 */

        /* if statement */
        if (month==1) {
            i=1;
        }
    }
```
/* nested if statement */
if (month==1) {
   if (day==1) {
      z=1;
   } else { /* (day==1) */
      z=2;
   }
}
/* end (month==1) */

/* for loop */
j=0;
for (int k=1;k<10;k++) {
   j=j+1;
}

/* while loop */
while (j != -1) {
   z=z+1;
}

/* nested while loop */
while (j != -1) {
   for (int m=1;m<10;m++)
      z=z+1;
}

/* single line code change */
j = READDB(‘FIRST’);
/* while (j != -1) { delete : 19991114 cr1348 RD123 */
while (j != gk_ok {
   /* insert : 19991114 cr1348 RD123 */
   for (int m=1;m<10;m++)
      z=z+1;
}
   j = READDB(’NEXT’);
}

/* nested Loops */
/* while (j < 5) { start delete : 19991114 cr1350 RD182 */
/* for (int k2=1;k2<26;k2++) { */
/* if (intArrray[k2*2]==1) { */
/* lv_colNotEmptyAt = k2*2; 19991114 cr1348 RD123 */
/* } */
/* } */
/* end delete : 19991114 cr1350 RD182 */
while (j <= intArrrayCols ) {
    for (int k2=1;k2<intArrrayRows;k2++) {
        if (intArrray[k2*2]==1) {
            lv_colNotEmptyNext = k2*2-1;
        }
    }
} /* end insert : 19991114 cr1350 RD182 */

/* case statement */
switch (month) {
    case 1:
        System.out.println("January");
        break;
    case 2:
        System.out.println("February");
        break;
    case 3:
        System.out.println("March");
        break;
    case 4:
        System.out.println("April");
        break;
    case 5:
        System.out.println("May");
        break;
    case 6:
        System.out.println("June");
        break;
    case 7:
        System.out.println("July");
}
break;
case 8:
    System.out.println("August");
    break;
case 9:
    System.out.println("September");
    break;
case 10:
    System.out.println("October");
    break;
case 11:
    System.out.println("November");
    break;
default:
    System.out.println("December");
    break;
}

return;
Development as a Source of Documentation
AcE Home Owners Freedom Insurance System
System Requirements

1 Table of Contents

2 Introduction

This document will serve as a reference for the requirements of the Home Owners Freedom insurance system (HOF). This document along with the documents referenced by it will provide a complete description of the functions HOF will perform.

3 Visible Characteristics

This section describes the characteristics of the system that are visible to the user. Predominant, therefore, is the user interface although reports will also be described here.

3.1 Panel Definitions
**pl-32-4 Coordinates:**

<table>
<thead>
<tr>
<th>X Coordinates Top</th>
<th>X Coordinates Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_T^1 = x_{BL} + 42.4$ pt.</td>
<td>$x_{BL} = G1HF325E.BLX_{eq}$</td>
</tr>
<tr>
<td>$x_T^2 = x_{BL} + 110.8$ pt.</td>
<td>$x_B = x_{BL} + 39.5$ pt.</td>
</tr>
<tr>
<td>$x_T^3 = x_{BL} + 119.3$ pt.</td>
<td>$x_B^3 = x_{BL} + 87.3$ pt.</td>
</tr>
<tr>
<td>$x_T^4 = x_{BL} + 188.5$ pt.</td>
<td>$x_B^4 = x_{BL} + 122.2$ pt.</td>
</tr>
<tr>
<td>$x_T^5 = x_{BL} + 430.8$ pt.</td>
<td>$x_B^5 = x_{BL} + 204.9$ pt.</td>
</tr>
<tr>
<td></td>
<td>$x_B^6 = x_{BL} + 231.9$ pt.</td>
</tr>
<tr>
<td></td>
<td>$x_B^7 = x_{BL} + 275.4$ pt.</td>
</tr>
<tr>
<td></td>
<td>$x_B^8 = x_{BL} + 318.9$ pt.</td>
</tr>
<tr>
<td></td>
<td>$x_B^9 = x_{BL} + 387.3$ pt.</td>
</tr>
<tr>
<td>$x_{TR} = x_{BL} + 493.5$ pt.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y Coordinates Left</th>
<th>Y Coordinates Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_{BL} = G1HF325E.BLY_{eq}$</td>
<td>$y_{TR} = x_{BL} + 515.1$ pt.</td>
</tr>
<tr>
<td>$y_{L}^1 = y_{BL} + 63.8$ pt.</td>
<td>$y_R = y_{BL} + 119.4$ pt.</td>
</tr>
<tr>
<td>$y_{L}^2 = y_{BL} + 180$ pt.</td>
<td>$y_R^2 = y_{BL} + 174.3$ pt.</td>
</tr>
<tr>
<td>$y_{L}^3 = y_{BL} + 227.7$ pt.</td>
<td>$y_R^3 = y_{BL} + 198.5$ pt.</td>
</tr>
<tr>
<td>$y_{L}^4 = y_{BL} + 260.5$ pt.</td>
<td>$y_R^4 = y_{BL} + 219.9$ pt.</td>
</tr>
<tr>
<td>$y_{L}^5 = y_{BL} + 284$ pt.</td>
<td>$y_R^5 = y_{BL} + 244.8$ pt.</td>
</tr>
<tr>
<td>$y_{L}^6 = y_{BL} + 318.9$ pt.</td>
<td>$y_R^6 = y_{BL} + 474.5$ pt.</td>
</tr>
<tr>
<td>$y_{L}^7 = y_{BL} + 404.4$ pt.</td>
<td>$y_R^7 = y_{BL} + 486.6$ pt.</td>
</tr>
<tr>
<td>$y_{L}^8 = y_{BL} + 425.2$ pt.</td>
<td></td>
</tr>
<tr>
<td>$y_{L}^9 = y_{BL} + 458$ pt.</td>
<td></td>
</tr>
<tr>
<td>$y_{L}^{10} = y_{BL} + 484.3$ pt.</td>
<td></td>
</tr>
</tbody>
</table>

**pl-32-5 Crossreference:**

<table>
<thead>
<tr>
<th>VBB or Image</th>
<th>Instance</th>
<th>instance:BLX</th>
<th>instance:BLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;$G1HG325E&quot;</td>
<td>slD</td>
<td>:BLX_{eq} = x_T^1</td>
<td>:BLY_{eq} = y_R^1</td>
</tr>
<tr>
<td>countrySel_{obb}</td>
<td>riskCountry</td>
<td>:BLX_{eq} = x_T^2</td>
<td>:BLY_{eq} = y_R^2</td>
</tr>
<tr>
<td>address_{obb}</td>
<td>riskAddress</td>
<td>:BLX_{eq} = x_T^3</td>
<td>:BLY_{eq} = y_R^3</td>
</tr>
<tr>
<td>checkBoxVirtCal_{obb}</td>
<td>riskUorR</td>
<td>:BLX_{eq} = x_T^4</td>
<td>:BLY_{eq} = y_R^4</td>
</tr>
<tr>
<td>distanceSel_{obb}</td>
<td>riskHydrantDist</td>
<td>:BLX_{eq} = x_T^5</td>
<td>:BLY_{eq} = y_R^5</td>
</tr>
</tbody>
</table>

Print Date: September 26, 2001 V02-R11 Home Owner Freedom System
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primaryUse_{rb}</code></td>
<td>Risk primary use</td>
</tr>
<tr>
<td><code>dateTime_{rb}</code></td>
<td>Risk on risk start</td>
</tr>
<tr>
<td><code>dateTime_{rb}</code></td>
<td>Risk on risk end</td>
</tr>
<tr>
<td><code>sumInsured_{rb}</code></td>
<td>Risk sum insured</td>
</tr>
<tr>
<td><code>messageToUser_{rb}</code></td>
<td>Message to user</td>
</tr>
<tr>
<td><code>nextButton_{rb}</code></td>
<td>Next button</td>
</tr>
<tr>
<td><code>exitButton_{rb}</code></td>
<td>Exit button</td>
</tr>
<tr>
<td><code>saveButton_{rb}</code></td>
<td>Save button</td>
</tr>
<tr>
<td><code>aceIcon_{im}</code></td>
<td>Literally heading G1HF24EN_{im}</td>
</tr>
<tr>
<td><code>litHOFEN_{im}</code></td>
<td>Literally heading risk 24EN_{im}</td>
</tr>
<tr>
<td><code>litCountryEN_{im}</code></td>
<td>Literally country EN_{im}</td>
</tr>
<tr>
<td><code>litUrbanEN_{im}</code></td>
<td>Literally urban EN_{im}</td>
</tr>
<tr>
<td><code>litRuralEN_{im}</code></td>
<td>Literally rural EN_{im}</td>
</tr>
<tr>
<td><code>litHydrantDistEN_{im}</code></td>
<td>Literally hydrant dist EN_{im}</td>
</tr>
<tr>
<td><code>litPrimaryUseEN_{im}</code></td>
<td>Literally primary use EN_{im}</td>
</tr>
<tr>
<td><code>litOnRiskPeriodEN_{im}</code></td>
<td>Literally on risk period EN_{im}</td>
</tr>
<tr>
<td><code>litStartDateTimeEN_{im}</code></td>
<td>Literally start date time EN_{im}</td>
</tr>
<tr>
<td><code>litEndDateTimeEN_{im}</code></td>
<td>Literally end date time EN_{im}</td>
</tr>
<tr>
<td><code>litSumInsuredEN_{im}</code></td>
<td>Literally sum insured EN_{im}</td>
</tr>
<tr>
<td><code>litRegionEN_{im}</code></td>
<td>Literally region EN_{im}</td>
</tr>
<tr>
<td><code>litCityEN_{im}</code></td>
<td>Literally city EN_{im}</td>
</tr>
<tr>
<td><code>litRiskAddressEN_{im}</code></td>
<td>Literally risk address EN_{im}</td>
</tr>
</tbody>
</table>

### Controlled Quantities

#### cq-32.1

**Top left corner of the visual display area**

- **Identifier**: `BLX_{cq}`
- **Description**: `G1HF325E.BLX_{cq}`
- **Hardware**: n.a.

---

3These images have not been included in the case study but would be defined in the same way as `literalRiskAddressDE_{im}`.

---

Print Date: September 26, 2001          V02-R11          Home Owner Freedom System
Top left corner of the visual display area

Identifier: $\text{BLY}_{cq}$

Description: $\text{BLX}_{cq}$

Hardware: n.a.

Bottom right corner of the visual display area

Identifier: $\text{TRX}_{cq}$

Description: $\text{BLY}_{cq} = \text{BLX} + !!x_{br}!!$

Hardware: n.a.

Bottom right corner of the visual display area

Identifier: $\text{TRY}_{cq}$

Description: $\text{BLY}_{cq} = \text{BLX} + !!x_{by}!!$

Hardware: n.a.

Visual display area

Identifier: $\text{visualDisplayArea}_{cq}$

Description: $\text{displayArea}_{cq} = \text{g1hf325e}_{pl}$

Hardware: The portion of the screen within the rectangle described by $\text{BLX}_{cq}$, $\text{BLY}_{cq}$ and $\text{BLX}_{cq}$ and $\text{BLY}_{cq}$.

Messages to the user

Identifier: $\text{messageToUser}_{cq}$

Description: $\text{uEclassCode}_{cq} = \text{hfMessageToUser.uEclassCode}_{ov}$

G1HF325E.messageToUser.uELctnCode$_{cq} = \text{hfUeLctnCode}_{ov}$

G1HF325E.messageToUser.text$_{cq} = \text{hfMessageToUser.text}_{ov}$
cq-32.6-3  **Hardware:**
  n.a.
pl-32  **Monitored Quantities**

mq-32.1  **Risk Country**
mq-32.1-1  **Identifier:**  riskCountry
mq-32.1-2  **Description:**
  Capture of risk country in English language.
mq-32.1-3  **Hardware/Software:**
  $g1hf325e.riskCountry = riskCountry; seln_{mq}$

mq-32.2  **Risk address**
mq-32.2-1  **Identifier:**  riskAddress
mq-32.2-2  **Description:**
  Capture of risk address in English language.
mq-32.2-3  **Hardware/Software:**
  $G1HF325E.RiskAddressText1_{mq} = riskAddress; text_{mq}[1]$
  $G1HF325E.RiskAddressText2_{mq} = riskAddress; text_{mq}[2]$
  $G1HF325E.RiskAddressText3_{mq} = riskAddress; text_{mq}[3]$
  $G1HF325E.RiskAddressCity_{mq} = riskAddress; city_{mq}$
  $G1HF325E.RiskAddressRegion_{mq} = riskAddress; region_{mq}$
  $G1HF325E.RiskAddressPostCode_{mq} = riskAddress; postCode_{mq}$

mq-32.3  **Urban or rural property.**
mq-32.3-1  **Identifier:**  setting
mq-32.3-2  **Description:**
  Urban or rural selection for risk property.
mq-32.3-3  **Hardware/Software:**
  $G1HF325E.setting_{mq} = \text{urban} \text{\$ WHEN riskUorR; check}_{mq} = \$1\text\$ otherwise$ $G1HF325E.setting_{mq} = \text{rural}\$\$

mq-32.4  **Distance to hydrant**
mq-32.4-1  **Identifier:**  nearestHydrant

---

4These images have not been included in the case study but would be defined in the same way as literalRiskAdressDEim.
mq-32.4-2 **Description:**
Distance to nearest hydrant for rural properties.

**Hardware/Software:**

| $G1HF325E.\text{nearestHydrantDistance}_{mq}$ |
| $\text{riskHydrantDistance}_{cq}$ |
| WHEN $G1HF325E.\text{setting}_{mq}$ |
| $\text{rural}$ |
| $G1HF325E.\text{nearestHydrantUnits}_{mq}$ |
| $G1HF325E.\text{nearestHydrantData0}_{mq}$ |
| $\text{true}$ |
| $G1HF325E.\text{nearestHydrantDistance}_{mq}$ |
| $\text{zero}_{ck}$ |
| $G1HF325E.\text{nearestHydrantUnits}_{mq}$ |
| $\text{blank}_{ck}$ |
| $G1HF325E.\text{nearestHydrantData0}_{mq}$ |
| $\text{false}$ |
| otherwise |

mq-32.5 **Primary use**

**Identifier:** $\text{primaryUse}_{mq}$

**Description:** The insured objects primary use.

**Hardware/Software:**

$G1HF325E.\text{primaryUse}_{mq} = \text{riskPrimaryUse:choice}_{cq}$

mq-32.6 **Start of risk period**

**Identifier:** $\text{onRiskStart}_{mq}$

**Description:** The date and time starting from which the risk is insured.

**Hardware/Software:**

$G1HF325E.\text{onRiskStartDate}_{mq} = \text{riskOnRiskStart:date}_{cq}$

$G1HF325E.\text{onRiskStartTime}_{mq} = \text{riskOnRiskStart:time}_{cq}$

mq-32.7 **End of risk period**

**Identifier:** $\text{onRiskEnd}_{mq}$

**Description:** The date and time at which the risk is no longer insured.

**Hardware/Software:**
G1HF325E.onRiskEndDate_{mq} = riskOnRiskEnd:date_{cq}
G1HF325E.onRiskEndTime_{mq} = riskOnRiskEnd:time_{cq}

<table>
<thead>
<tr>
<th>mq-32.8</th>
<th><strong>Sum insured</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>mq-32.8-1</td>
<td><strong>Identifier:</strong> sumInsured_{mq}</td>
</tr>
<tr>
<td>mq-32.8-2</td>
<td><strong>Description:</strong> The value of the insured risk.</td>
</tr>
<tr>
<td>mq-32.8-3</td>
<td><strong>Hardware/Software:</strong></td>
</tr>
<tr>
<td></td>
<td>G1HF325E.sumInsuredValue_{mq} = riskSumInsured:value_{cq}</td>
</tr>
<tr>
<td></td>
<td>G1HF325E.sumInsuredCurrency_{mq} = riskSumInsured:currency_{cq}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mq-32.9</th>
<th><strong>Next button</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>mq-32.9-1</td>
<td><strong>Identifier:</strong> nextButton_{mq}</td>
</tr>
<tr>
<td>mq-32.9-2</td>
<td><strong>Description:</strong> The next button.</td>
</tr>
<tr>
<td>mq-32.9-3</td>
<td><strong>Hardware/Software:</strong></td>
</tr>
<tr>
<td></td>
<td>G1HF325E.nextButton_{mq} = nextButton:actuationState_{cq}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mq-32.10</th>
<th><strong>Save button</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>mq-32.10-1</td>
<td><strong>Identifier:</strong> saveButton_{mq}</td>
</tr>
<tr>
<td>mq-32.10-2</td>
<td><strong>Description:</strong> The save button.</td>
</tr>
<tr>
<td>mq-32.10-3</td>
<td><strong>Hardware/Software:</strong></td>
</tr>
<tr>
<td></td>
<td>G1HF325E.saveButton_{mq} = saveButton:push_{cq}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mq-32.11</th>
<th><strong>Exit button</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>mq-32.11-1</td>
<td><strong>Identifier:</strong> exitButton_{mq}</td>
</tr>
<tr>
<td>mq-32.11-2</td>
<td><strong>Description:</strong> The exit button.</td>
</tr>
<tr>
<td>mq-32.11-3</td>
<td><strong>Hardware/Software:</strong></td>
</tr>
<tr>
<td></td>
<td>G1HF325E.exitButton_{mq} = exitButton:push_{cq}</td>
</tr>
</tbody>
</table>
pl-32  

**Modes Class**

### mc-32.1-1
**Identifier:**  g1hf325e

### mc-32.1-2
**Description:**  Text1 modes.

### mc-32.1-3
**Initial Mode:**  riskCountryDE

### mc-32.1-4
**Transitions:**

<table>
<thead>
<tr>
<th>From Mode</th>
<th>To Mode</th>
<th>T01</th>
<th>T02</th>
<th>T03</th>
<th>T04</th>
<th>T05</th>
<th>T06</th>
<th>T07</th>
<th>T08</th>
<th>T09</th>
<th>T10</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1hf325eCountryDtEn*mds</td>
<td>g1hf325eCountryDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eAddressDtEn*mds</td>
<td>g1hf325eAddressDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eUorRDtEn*mds</td>
<td>g1hf325eUorRDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eNearestHydrantDtEn*mds</td>
<td>g1hf325eNearestHydrantDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325ePrimaryUseDtEn*mds</td>
<td>g1hf325ePrimaryUseDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eStartDtEn*mds</td>
<td>g1hf325eStartDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eEndDtEn*mds</td>
<td>g1hf325eEndDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eSumInsuredDtEn*mds</td>
<td>g1hf325eSumInsuredDtEn*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eBttnNext*mds</td>
<td>g1hf325eBttnNext*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eBttnSave*mds</td>
<td>g1hf325eBttnSave*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325eSumExit*mds</td>
<td>g1hf325eSumExit*mds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### mc-32.1-5
**Events:**

### ev-32.1-1
**Identifier:**  T01

### ev-32.1-2
**From:**  g1hf325eCountryDtEn*mds

### ev-32.1-3
**To:**  g1hf325eaddressDtEn*mds

### ev-32.1-4
**Events:**

@T01(tab) WHEN G1HF325E.g1hf325eCurrentMode_eq = $riskCountryDtEn$
ev-32.2-1 Identifier: T02
From: g1hf325eaddressDtEn_{mds}
To: g1hf325eUorRDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskAddressDtEn$

ev-32.3-1 Identifier: T03
From: g1hf325eUorRDtEn_{mds}
To: g1hf325eNearestHydrantDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskUorRDtEn$

ev-32.4-1 Identifier: T04
From: g1hf325eNearestHydrantDtEn_{mds}
To: g1hf325eStartDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskHydrantDistDtEn$

ev-32.5-1 Identifier: T05
From: g1hf325eStartDtEn_{mds}
To: g1hf325eEndDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskStartDtEn$

ev-32.6-1 Identifier: T06
From: g1hf325eEndDtEn_{mds}
To: g1hf325ePrimeryUseDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskEndDtEn$

ev-32.7-1 Identifier: T07
From: g1hf325ePrimeryUseDtEn_{mds}
To: g1hf325eSumInsuredDtEn_{mds}
Events: @T_{i0}(tab) WHEN G1HF325E.g1hf325eCurrentMode\_{cq} = $riskPrimeryUseDtEn$
ev-32.8-1  **Identifier:** T08
ev-32.8-2  **From:** g1hf325eSumInsuredDtlEn
           **To:** g1hf325eBtnNext
ev-32.8-4  **Events:** @T08(tab) WHEN G1HF325E.g1hf325eCurrentMode
            = $riskSumInsuredDtlEn$

ev-32.9-1  **Identifier:** T09
ev-32.9-2  **From:** g1hf325eBtnNext
           **To:** g1hf325eBtnSave
ev-32.9-4  **Events:** @T08(tab) WHEN G1HF325E.g1hf325eCurrentMode
            = $riskBtnNextFsOn$

ev-32.10-1 **Identifier:** T10
ev-32.10-2 **From:** g1hf325eBtnSave
           **To:** g1hf325eSumExit
ev-32.10-4 **Events:** @T08(tab) WHEN G1HF325E.g1hf325eCurrentMode
            = $riskBtnSaveFsOn$

ev-32.11-1 **Identifier:** T11
ev-32.11-2 **From:** g1hf325eSumExit
           **To:** g1hf325eCountryDtlEn
ev-32.11-4 **Events:** @T08(tab) WHEN G1HF325E.g1hf325eCurrentMode
            = $riskBtnExitFsOn$

pl-32  **Modes**
See Address VBB for an example

pl-32  **Demand Functions**
  **df-32.1-1**  **Identifier:** pushButton
  **df-32.1-2**  **Modes:** g1hf325eBtnNextFsOn
                 g1hf325eBtnSaveFsOn
                 g1hf325eBtnExitFsOn
  **df-32.1-3**  **Request and Display Description:**
SR-12

Visible Characteristics

**g1hf325e**

<table>
<thead>
<tr>
<th>Modes</th>
<th>Events</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1hf325eBtnNextFsOnly&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(enter)</td>
<td>g1hfButtonActuated&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eBtnNext $</td>
</tr>
<tr>
<td>g1hf325eBtnSaveFsOnly&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(enter)</td>
<td>g1hfButtonActuated&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eSave $</td>
</tr>
<tr>
<td>g1hf325eBtnExitFsOnly&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(enter)</td>
<td>g1hfButtonActuated&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eExit $</td>
</tr>
</tbody>
</table>

**df-32.2.1** **Identifier:** pushButton

**df-32.2.2** **Modes:**
g1hf325eBtnNext<sub>m</sub>, g1hf325eBtnSave<sub>m</sub>,
g1hf325eCountryDtEn<sub>m</sub>,
g1hf325eEndDtEn<sub>m</sub>,
g1hf325eNearestHydrantDtEn<sub>m</sub>,
g1hf325ePrimaryUseDtEn<sub>m</sub>,
g1hf325eStartDtEn<sub>m</sub>,
g1hf325eSumExit<sub>m</sub>,
g1hf325eSumInsuredDtEn<sub>m</sub>,
g1hf325eUorRdDtEn<sub>m</sub>,
g1hf325eaddressDtEn<sub>m</sub>

**df-32.2.3** **Request and Display Description:**

<table>
<thead>
<tr>
<th>Modes</th>
<th>Events</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1hf325eCountryDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eCountryDtEn $</td>
</tr>
<tr>
<td>g1hf325eaddressDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eaddressDtEn $</td>
</tr>
<tr>
<td>g1hf325eUorRdDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eNearestHydrantDtEn $</td>
</tr>
<tr>
<td>g1hf325eNearestHydrantDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325ePrimaryUseDtEn $</td>
</tr>
<tr>
<td>g1hf325ePrimaryUseDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eStartDtEn $</td>
</tr>
<tr>
<td>g1hf325eStartDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eEndDtEn $</td>
</tr>
<tr>
<td>g1hf325eEndDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eSumInsuredDtEn $</td>
</tr>
<tr>
<td>g1hf325eSumInsuredDtEn&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eBtnNext $</td>
</tr>
<tr>
<td>g1hf325eBtnSave&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eBtnSave $</td>
</tr>
<tr>
<td>g1hf325eBtnExit&lt;sub&gt;m&lt;/sub&gt;</td>
<td>@T&lt;sub&gt;10&lt;/sub&gt;(tab)</td>
<td>g1hf325eCurrentMode&lt;sub&gt;eq&lt;/sub&gt; :=g1hf325eBtnExit $</td>
</tr>
</tbody>
</table>
pl-32  **Periodic Functions**
No periodic functions.

More Panels would go here
3.2 Visual Building Block Definitions

vbb-1
Property Address

Identifier: address_{vbb}

Description:
Visual building block for a property address that can be used as a mailing address or as a basis for a location address.

vbb-1.3
Image Layout:

vbb-1.3.3
Co-ordinates:

<table>
<thead>
<tr>
<th>X Coordinates</th>
<th>Y Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{bl} = \text{text30:BLX}_{eq} )</td>
<td>( y_{bl} = \text{text30:BLY}_{eq} )</td>
</tr>
<tr>
<td>( x^1 = x_{bl} + 4.2 ) pt</td>
<td>( y^1 = y_{bl} + 5.5 ) pt</td>
</tr>
<tr>
<td>( x^2 = x_{bl} + 183.1 ) pt</td>
<td>( y^2 = y_{bl} + 40.8 ) pt</td>
</tr>
<tr>
<td>( x^3 = x_{bl} + 189.5 ) pt</td>
<td>( y^3 = y_{bl} + 75.4 ) pt</td>
</tr>
<tr>
<td>( x_{tr} = x_{bl} + 258.4 ) pt</td>
<td>( y_{tr} = y_{bl} + 100.5 ) pt</td>
</tr>
<tr>
<td>( y_{bl} )</td>
<td>( y_{tr} = y_{bl} + 125.4 ) pt</td>
</tr>
</tbody>
</table>
vbb-1.3.4 Cross Reference:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Visual Element</th>
<th>( \text{instance: BLX} = x )</th>
<th>( \text{instance: BLY} = y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>text1</td>
<td>text\textsubscript{30} \textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{text} = x^1 )</td>
<td>( \text{instance: BLY}_{text} = y^1 )</td>
</tr>
<tr>
<td>text2</td>
<td>text\textsubscript{30} \textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{text} = x^1 )</td>
<td>( \text{instance: BLY}_{text} = y^2 )</td>
</tr>
<tr>
<td>text3</td>
<td>text\textsubscript{30} \textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{text} = x^1 )</td>
<td>( \text{instance: BLY}_{text} = y^3 )</td>
</tr>
<tr>
<td>city</td>
<td>text\textsubscript{15} \textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{city} = x^1 )</td>
<td>( \text{instance: BLY}_{city} = y^4 )</td>
</tr>
<tr>
<td>region</td>
<td>selBox\textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{region} = x^2 )</td>
<td>( \text{instance: BLY}_{region} = y^5 )</td>
</tr>
<tr>
<td>postCode</td>
<td>postCode\textsubscript{vbb}</td>
<td>( \text{instance: BLX}_{postCode} = x^3 )</td>
<td>( \text{instance: BLY}_{postCode} = y^6 )</td>
</tr>
</tbody>
</table>

vbb-1-4 Controlled Quantities

cq-0

Top left corner of the visual display area

**Identifier:** \( \text{BLX}_{cq} \)

**Description:**
\( \text{address.BLX}_{cq} = x \) coordinate of top left corner of Visual display area.

**Hardware:**
- n.a.

cq-1

Top left corner of the visual display area

**Identifier:** \( \text{BLY}_{cq} \)

**Description:**
\( \text{address.BLY}_{cq} = y \) coordinate of top left corner of Visual display area.

**Hardware:**
- n.a.

cq-2

Bottom right corner of the visual display area

**Identifier:** \( \text{TRX}_{cq} \)

**Description:**
\( \text{address.TRX}_{cq} = \text{address.BLX}_{cq} + \| x_{br} \| \).

**Hardware:**
- n.a.

cq-3

Bottom right corner of the visual display area

**Identifier:** \( \text{TRY}_{cq} \)

**Description:**
\( \text{address.TRY}_{cq} = \text{address.BLY}_{cq} + \| x_{by} \| \).

Print Date: September 26, 2001  V02-R11  Home Owner Freedom System
cq-3-3  **Hardware:**
n.a.

cq-4  **Visual display area**

  cq-4-1  **Identifier:** visualDisplayArea_{cq}

  cq-4-2  **Description:**
  
  \[
  \text{address.displayArea}_{cq} = \{ \text{text30Dormant}_{im}, \text{text30DataEntry}_{im}, \text{text30DataEntry}_{im} \}
  \]

  cq-4-3  **Hardware:**
  
  The portion of the screen within the rectangle described by address.BLX_{cq}, address.BLY_{cq} and address.BLX_{cq} and address.BLY_{cq}.

cq-5  **Region Selection options**

  cq-5-1  **Identifier:** regionSel_{cq}

  cq-5-2  **Description:**
  
  \[
  \text{region:selectionList}_{cq} = \text{cdnProvinces}_{ck}, \text{gbrShires}_{ck}, \text{deuStates}_{ck}
  \]

  cq-5-3  **Hardware:**
  
  n.a.

vbb-1.5  **Monitored Quantities**

mq-1  **Free form text portion of a formatted address.**

  mq-1-1  **Identifier:** addressText_{mq}

  mq-1-2  **Description:**
  
  \[
  \text{address.text}_{mq}[1] = \text{text1:KeyedText}_{mq} \\
  \text{address.text}_{mq}[2] = \text{text2:KeyedText}_{mq} \\
  \text{address.text}_{mq}[3] = \text{text3:KeyedText}_{mq}
  \]

  mq-1-3  **Hardware/Software:**
  
  VBB text30

mq-2  **City portion of a formatted address.**

  mq-2-1  **Identifier:** city_{mq}

  mq-2-2  **Description:**
  
  \[
  \text{address.city}_{mq} = \text{city:KeyedText}_{mq}
  \]

  mq-2-3  **Hardware/Software:**
  
  VBB text30
VBB text 15

mq-3 Region in which the city is located.
  mq-3-1 Identifier: region\textsubscript{mq}
  mq-3-2 Description: address.region\textsubscript{mq} = region:selection\textsubscript{mq}
  mq-3-3 Hardware/Software: VBB selBox

mq-4 postal code for the address.
  mq-4-1 Identifier: postCode\textsubscript{mq}
  mq-4-2 Description: address.postCode\textsubscript{mq} = postCode:enteredCode\textsubscript{mq}
  mq-4-3 Hardware/Software: VBB postCode

vbb-1.6 Modes Class

mc-1-1 Identifier: text30
mc-1-2 Description: Text1 modes.
  mc-1-3 Initial Mode: defined by initiating VBB or panel
  mc-1-4 Transitions:
Events:

**ev-1-1**  
**Identifier:** T01  
**From:** addressT1DataEntry<sub>mde</sub>  
**To:** addressT2DataEntry<sub>mde</sub>  
**Events:**  
@T<sub>01</sub>(tab) WHEN address.addressCurrentlyMode<sub>eq</sub> = $addressT1DataEntry$  

**ev-2-1**  
**Identifier:** T02  
**From:** addressT2DataEntry<sub>mde</sub>  
**To:** addressT3DataEntry<sub>mde</sub>  
**Events:**  
@T<sub>02</sub>(tab) WHEN address.addressCurrentlyMode<sub>eq</sub> = $addressT2DataEntry$  

**ev-3-1**  
**Identifier:** T03  
**From:** addressT3DataEntry<sub>mde</sub>  
**To:** addressCityDataEntry<sub>mde</sub>  
**Events:**
Condition:
address.addressCurrentMode\textsubscript{cq} = $\text{addressT1DataEntry}$

Identifier: addressT2DataEntry
Description: Data entry into the second text30\textsubscript{vb}. Values:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address.visualDisplayArea\textsubscript{cq} :address\textsubscript{sn}</td>
<td></td>
</tr>
<tr>
<td>addressText\textsubscript{mq}[2] :text\textsubscript{2}:KeyedText\textsubscript{mq}</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{1}:address.text30CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{2}:address.text30CurrentMode\textsubscript{cq} :$\text{dataEntry}$</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{3}:address.text30CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>city:address.text15CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>region:address.text15CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>postCode:address.postCodeCurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
</tbody>
</table>

Condition:
address.addressCurrentMode\textsubscript{cq} = $\text{addressT2DataEntry}$

Identifier: addressT3DataEntry
Description: Data entry into the third text30\textsubscript{vb}. Values:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address.visualDisplayArea\textsubscript{cq} :address\textsubscript{sn}</td>
<td></td>
</tr>
<tr>
<td>addressText\textsubscript{mq}[3] :text\textsubscript{3}:KeyedText\textsubscript{mq}</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{1}:address.text30CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{2}:address.text30CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>text\textsubscript{3}:address.text30CurrentMode\textsubscript{cq} :$\text{dataEntry}$</td>
<td></td>
</tr>
<tr>
<td>city:address.text15CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>region:address.text15CurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
<tr>
<td>postCode:address.postCodeCurrentMode\textsubscript{cq} :$\text{dormant}$</td>
<td></td>
</tr>
</tbody>
</table>

Condition:
address.addressCurrentMode\textsubscript{cq} = $\text{addressT2DataEntry}$

Identifier: addressCityDataEntry
Description: Data entry into the city text30\textsubscript{vb}. Values:
address.visualDisplayArea<sub>cq</sub> : <code>address<sub>sm</sub></code>
addressCityText<sub>mq</sub> : <code>city:KeyedText<sub>mq</sub></code>
text1:address.text30CurrentMode<sub>cq</sub> : $dormant$
text2:address.text30CurrentMode<sub>cq</sub> : $dormant$
text3:address.text30CurrentMode<sub>cq</sub> : $dormant$
city:address.text15CurrentMode<sub>cq</sub> : $dataEntry$
region:address.text15CurrentMode<sub>cq</sub> : $dormant$
postCode:address.postCodeCurrentMode<sub>cq</sub> : $dormant$

**Condition:**
address.addressCurrentMode<sub>cq</sub> = $addressCityDataEntry$

**Identifier:** addressRegionDataEntry

**Description:**
Data entry into the region text30<sub>cqb</sub>.

**Values:**
address.visualDisplayArea<sub>cq</sub> : <code>address<sub>sm</sub></code>
addressRegionText<sub>mq</sub> : <code>region:KeyedText<sub>mq</sub></code>
text1:address.text30CurrentMode<sub>cq</sub> : $dormant$
text2:address.text30CurrentMode<sub>cq</sub> : $dormant$
text3:address.text30CurrentMode<sub>cq</sub> : $dormant$
city:address.text15CurrentMode<sub>cq</sub> : $dataEntry$
region:address.text15CurrentMode<sub>cq</sub> : $dormant$
postCode:address.postCodeCurrentMode<sub>cq</sub> : $dormant$

**Condition:**
address.addressCurrentMode<sub>cq</sub> = $addressRegionDataEntry$

**Identifier:** addressPostCodeDataEntry

**Description:**
Data entry into the postCode text30<sub>cqb</sub>.

**Values:**
address.visualDisplayAreacq :address_sm
addressPostCodeTextmq :postCode:KeyedTextmq
text1:address.text30CurrentModecq,:dormant$
text2:address.text30CurrentModecq,:dormant$
text3:address.text30CurrentModecq,:dormant$
city:address.text15CurrentModecq,:dormant$
region:address.text15CurrentModecq,:dormant$
postCode:address.postCodeCurrentModecq,:dataEntry$

md-6-4 Condition:
address.addressCurrentModecq = $addressPostCodeDataEntry$

md-7-1 Identifier: addressDormant
md-7-2 Description: All focus is off address_vbb.
md-7-3 Values:
address.visualDisplayAreacq :address_sm
text1:address.text30CurrentModecq,:dormant$
text2:address.text30CurrentModecq,:dormant$
text3:address.text30CurrentModecq,:dormant$
city:address.text15CurrentModecq,:dormant$
region:address.text15CurrentModecq,:dormant$
postCode:address.postCodeCurrentModecq,:dormant$

md-7-4 Condition:
address.addressCurrentModecq = $addressDormant$

vbb-1.8 Demand Functions
df-1-1 Identifier: addressTab
df-1-2 Modes: text1DataEntry*mds
df-1-3 Request and Display Description:
## Periodic Functions

No periodic functions.

### Visible Characteristics

```
<table>
<thead>
<tr>
<th>Modes</th>
<th>Events</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressT1DataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressT2DataEntry$_{eq} :=</td>
</tr>
<tr>
<td>addressT2DataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressT3DataEntry$_{eq} :=</td>
</tr>
<tr>
<td>addressT3DataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressCityDataEntry$_{eq} :=</td>
</tr>
<tr>
<td>addressCityDataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressRegionDataEntry$_{eq} :=</td>
</tr>
<tr>
<td>addressRegionDataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressPostCodeDataEntry$_{eq} :=</td>
</tr>
<tr>
<td>addressPostCodeDataEntry_{md}s</td>
<td>@T_{10}(tab)</td>
<td>address.addressCurrentMode_{eq} :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$addressDormant$_{eq} :=</td>
</tr>
</tbody>
</table>
```
Thirty Character Text

**Identifier:** Text30

**Description:**
Visual building block for a thirty character data input field.

**Image Layout:**

**Co-ordinates:**

<table>
<thead>
<tr>
<th>X Coordinates Top</th>
<th>X Coordinates Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{text30:ELx}_{ol} )</td>
<td>( x_{16} = x_{BL} + 192\text{pt.} )</td>
</tr>
<tr>
<td>( x^1 = x_{BL} + 12\text{pt.} )</td>
<td>( x_{17} = x_{BL} + 204\text{pt.} )</td>
</tr>
<tr>
<td>( x^2 = x_{BL} + 24\text{pt.} )</td>
<td>( x_{18} = x_{BL} + 216\text{pt.} )</td>
</tr>
<tr>
<td>( x^3 = x_{BL} + 36\text{pt.} )</td>
<td>( x_{19} = x_{BL} + 228\text{pt.} )</td>
</tr>
<tr>
<td>( x^4 = x_{BL} + 48\text{pt.} )</td>
<td>( x_{20} = x_{BL} + 240\text{pt.} )</td>
</tr>
<tr>
<td>( x^5 = x_{BL} + 60\text{pt.} )</td>
<td>( x_{21} = x_{BL} + 252\text{pt.} )</td>
</tr>
<tr>
<td>( x^6 = x_{BL} + 72\text{pt.} )</td>
<td>( x_{22} = x_{BL} + 264\text{pt.} )</td>
</tr>
<tr>
<td>( x^7 = x_{BL} + 84\text{pt.} )</td>
<td>( x_{23} = x_{BL} + 276\text{pt.} )</td>
</tr>
<tr>
<td>( x^8 = x_{BL} + 96\text{pt.} )</td>
<td>( x_{24} = x_{BL} + 288\text{pt.} )</td>
</tr>
<tr>
<td>( x^9 = x_{BL} + 108\text{pt.} )</td>
<td>( x_{25} = x_{BL} + 300\text{pt.} )</td>
</tr>
<tr>
<td>( x^{10} = x_{BL} + 120\text{pt.} )</td>
<td>( x_{26} = x_{BL} + 312\text{pt.} )</td>
</tr>
<tr>
<td>( x^{11} = x_{BL} + 132\text{pt.} )</td>
<td>( x_{27} = x_{BL} + 324\text{pt.} )</td>
</tr>
<tr>
<td>( x^{12} = x_{BL} + 144\text{pt.} )</td>
<td>( x_{28} = x_{BL} + 336\text{pt.} )</td>
</tr>
<tr>
<td>( x^{13} = x_{BL} + 156\text{pt.} )</td>
<td>( x_{29} = x_{BL} + 348\text{pt.} )</td>
</tr>
<tr>
<td>( x^{14} = x_{BL} + 168\text{pt.} )</td>
<td>( x_{30} = x_{BL} + 360\text{pt.} )</td>
</tr>
<tr>
<td>( x^{15} = x_{BL} + 180\text{pt.} )</td>
<td>( x_{br} = x_{BL} + 372\text{pt.} )</td>
</tr>
</tbody>
</table>
vbb-2.3.4

**Cross Reference:**

<table>
<thead>
<tr>
<th>VBB or Image</th>
<th>Instance</th>
<th>instance:BLX</th>
<th>instance:BLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>text1_vbb</td>
<td>t1</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t2</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t3</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t4</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t5</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t6</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t7</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t8</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t9</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t10</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t11</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t12</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t13</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t14</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t15</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t16</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t17</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t18</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t19</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t20</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t21</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t22</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t23</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t24</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t25</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t26</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t27</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t28</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
<tr>
<td>text1_vbb</td>
<td>t29</td>
<td>:address.BLx_cq = x_BL _BL</td>
<td>:address.BLY_cq = y_BL</td>
</tr>
</tbody>
</table>
Controlled Quantities

**cq-2.1**

**Top left corner of the visual display area**

<table>
<thead>
<tr>
<th>Identifier:</th>
<th>BLX$_{cq}$</th>
</tr>
</thead>
</table>

**Description:**

BLX$_{cq}$ = x coordinate of top left corner of Visual display area.

**Hardware:**

n.a.

**cq-2.2**

**Top left corner of the visual display area**

<table>
<thead>
<tr>
<th>Identifier:</th>
<th>BLY$_{cq}$</th>
</tr>
</thead>
</table>

**Description:**

BLY$_{cq}$ = y coordinate of top left corner of Visual display area.

**Hardware:**

n.a.

**cq-2.3**

**Bottom right corner of the visual display area**

<table>
<thead>
<tr>
<th>Identifier:</th>
<th>TRX$_{cq}$</th>
</tr>
</thead>
</table>

**Description:**

TRX$_{cq}$ = !x$_{br}$ !.

**Hardware:**

n.a.

**cq-2.4**

**Bottom right corner of the visual display area**

<table>
<thead>
<tr>
<th>Identifier:</th>
<th>TRY$_{cq}$</th>
</tr>
</thead>
</table>

**Description:**

TRY$_{cq}$ = !x$_{by}$ !.

**Hardware:**

n.a.

**cq-2.5**

**Visual display area**

<table>
<thead>
<tr>
<th>Identifier:</th>
<th>visualDisplayArea$_{cq}$</th>
</tr>
</thead>
</table>

**Description:**

- $text1_{vbb} | t30 |
- $: address.BLX_{cq} = x_{BL}^{30}$ |
- $: address.BLY_{cq} = y_{BL}^{30}$ |
displayArea_{cq} = \{text30Dormant_{sm}, text30DataEntry_{sm},
    text30DataEntry_{sm}\}

cq-2.5-3  **Hardware:**
The portion of the screen within the rectangle described by
address.BL{cq}, address.BLY_{cq} and address.TR{cq} and address.TRY_{cq}.

cq-2.6  **Visual text**
cq-2.6-1  **Identifier:** visualText_{cq}
cq-2.6-2  **Description:**
address.visualText_{cq} = address.keyedText_{mq}
cq-2.6-3  **Hardware:**
keyed text character.

cq-2.7  **Mode the VBB is currently in.**
cq-2.7-1  **Identifier:** text1CurrentMode_{cq}
cq-2.7-2  **Description:**
address.text1CurrentMode_{cq} \in \{$dormant$, $dataEntry$\}
cq-2.7-3  **Hardware:**
n.a.

**Monitored Quantities**

**mq-2.1 Key entered text.**
mq-2.1-1  **Identifier:** inputText_{mq}
mq-2.1-2  **Description:**

\[
\begin{align*}
    \text{inputText}_{mq}[1] & = t1: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[2] & = t2: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[3] & = t3: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[4] & = t4: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[5] & = t5: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[6] & = t6: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[7] & = t7: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[8] & = t8: \text{keyedText}_{mq} \\
    \text{inputText}_{mq}[9] & = t9: \text{keyedText}_{mq}
\end{align*}
\]
inputText\_{mq,10} = t10: keyedText\_{mq}
inputText\_{mq,11} = t11: keyedText\_{mq}
inputText\_{mq,12} = t12: keyedText\_{mq}
inputText\_{mq,13} = t13: keyedText\_{mq}
inputText\_{mq,14} = t14: keyedText\_{mq}
inputText\_{mq,15} = t15: keyedText\_{mq}
inputText\_{mq,16} = t16: keyedText\_{mq}
inputText\_{mq,17} = t17: keyedText\_{mq}
inputText\_{mq,18} = t18: keyedText\_{mq}
inputText\_{mq,19} = t19: keyedText\_{mq}
inputText\_{mq,20} = t20: keyedText\_{mq}
inputText\_{mq,21} = t21: keyedText\_{mq}
inputText\_{mq,22} = t22: keyedText\_{mq}
inputText\_{mq,23} = t23: keyedText\_{mq}
inputText\_{mq,24} = t24: keyedText\_{mq}
inputText\_{mq,25} = t25: keyedText\_{mq}
inputText\_{mq,26} = t26: keyedText\_{mq}
inputText\_{mq,27} = t27: keyedText\_{mq}
inputText\_{mq,28} = t28: keyedText\_{mq}
inputText\_{mq,29} = t29: keyedText\_{mq}
inputText\_{mq,30} = t30: keyedText\_{mq}

**mq-2.1-3**  **Hardware/Software:**  
VBB text1

**vbb-2**  **Modes Class**

**mc-2.1-1**  **Identifier:**  text30

**mc-2.1-2**  **Description:**  
Text1 modes.

**mc-2.1-3**  **Initial Mode:**  defined by initiating VBB or panel

**mc-2.1-4**  **Transitions:**

Print Date: September 26, 2001  V02-R11  Home Owner Freedom System
### Events:

**ev-2.1-1**  
**Identifier:**  T30  
**From:**  text30DataEntry *mds*  
**To:**  text30Dormant *mds*  
**Events:**  
@T$_{01}$(tab) WHEN text30CurrentMode$_{aq} = $dataEntry$

**ev-2.2-1**  
**Identifier:**  K01  
**From:**  text30DataEntry *mds*  
**To:**  text30Dormant *mds*  
**Events:**  
@T$_{01}$(tab) WHEN text30CurrentMode$_{aq} = $dataEntry$
\[\land t30:currentMode_{aq} = $dataEntry$\]

### Modes

**md-2.1-1**  
**Identifier:**  text1DataEntry  
**Description:**  Data entry into text1_vbb.  
**Values:**

visualDisplayArea$_{aq}:text30DataEntry_{im}$  
inputText$_{mq}[1] : t1:keyedText$_{mq}$
Condition:
\[ \text{instance:asCQtext1CurrentMode} = \$\text{dataEntry}\$ \]

Identifier: text1Dormant

Description: No activity for text1_dbb.

Print Date: September 26, 2001
md-2.2-3  **Values:**
address.visualDisplayArea\textsubscript{eq} : text\textsuperscript{30Dormant}\textsubscript{im}

md-2.2-4  **Condition:**
instances: text\textsuperscript{1CurrentMode}\textsubscript{eq} \neq \$dataEntry\$

vbb-2  **Demand Functions**
No demand functions.

vbb-2  **Periodic Functions**
No periodic functions.
vbb-3  One Character Text
vbb-3-1  Identifier:  text1\_vbb
vbb-3-2  Description:
Visual building block for a single character data input field.
vbb-3-3  Image Layout:

![Image](image.png)

vbb-3-3.3  Co-ordinates:

<table>
<thead>
<tr>
<th>X Coordinates Top</th>
<th>X Coordinates Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{TR} = x_{BL} + 28 \text{ pt.} )</td>
<td>( x_{BL} = \text{text1.BLX}_{cq} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y Coordinates Left</th>
<th>Y Coordinates Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_{BL} = \text{text1.BLY}_{cq} )</td>
<td>( y_{TR} = y_{BL} + 14 \text{ pt.} )</td>
</tr>
</tbody>
</table>

vbb-3-4  Cross Reference:
none

vbb-3-4  Controlled Quantities

cq-3-4-1  Top left corner X
cq-3-4-1-1  Identifier:  BLX\_cq
cq-3-4-1-2  Description:
text1.BLX\_cq = The x coordinate of the upper left corner of an instance of a text1 image.
cq-3-4-1-3  Hardware:
n.a.

cq-3-4-2  Top left corner Y
cq-3-4-2-1  Identifier:  BLY\_cq
cq-3-4-2-2 **Description:**
\[
text1.BLY_{cq} = \text{The y coordinate of the upper left corner of an instance of a text1 image.}
\]

cq-3-4-2-3 **Hardware:**
\[
n.a.
\]

cq-3-4-3 **Bottom left corner X**

cq-3-4-3-1 **Identifier:** \(BLX_{cq}\)

cq-3-4-3-2 **Description:**
\[
text1.TRX_{cq} = ![x_{br}].
\]

cq-3-4-3-3 **Hardware:**
\[
n.a.
\]

cq-3-4-4 **Bottom left corner Y**

cq-3-4-4-1 **Identifier:** \(BLY_{cq}\)

cq-3-4-4-2 **Description:**
\[
text1.TRY_{cq} = ![y_{br}].
\]

cq-3-4-4-3 **Hardware:**
\[
n.a.
\]

cq-3-4-5 **Visual display area**

cq-3-4-5-1 **Identifier:** \(\text{visualDisplayArea}_{cq}\)

cq-3-4-5-2 **Description:**
\[
text1.displayArea_{cq} = \{\text{text1Dormant}_{im}, \text{text1DataEntry}_{im}\}
\]

cq-3-4-5-3 **Hardware:**
\[
\text{The portion of the screen within the rectangle described by text1.BLX}_{cq},
\text{text1.BLY}_{cq} \text{ and text1.BLY}_{cq} \text{ and text1.BLY}_{cq}.
\]

cq-3-4-6 **Visual text**

cq-3-4-6-1 **Identifier:** \(\text{visualText}_{cq}\)

cq-3-4-6-2 **Description:**
\[
text1.visualText_{cq} = text1.keyedText_{mq}
\]

cq-3-4-6-3 **Hardware:**
\[
\text{keyed text character.}
\]
cq-3-4-7 Mode the VBB is currently in.
cq-3-4-7-1 **Identifier:** CurrentMode<sub>cq</sub>
cq-3-4-7-2 **Description:**
text1.CurrentMode<sub>cq</sub> ∈ {$dormant$, $dataEntry$}
cq-3-4-7-3 **Hardware:**
n.a.

vbb-3-5 **Monitored Quantities**

mq-3-5-1 Key entered text.
mq-3-5-1-1 **Identifier:** keyedText<sub>mq</sub>
mq-3-5-1-2 **Description:**
text1.keyedText<sub>eq</sub> = instance:KeyStroke<sub>mq</sub>
mq-3-5-1-3 **Hardware/Software:**
Keyboard.

vbb-3-6 **Mode Class**

mc-3-6-1 **Identifier:** text1
mc-3-6-2 **Description:**
Text1 modes.
mc-3-6-3 **Initial Mode:** defined by initiating VBB or panel
mc-3-6-4 **Transitions:**

<table>
<thead>
<tr>
<th>From Mode</th>
<th>To Mode</th>
<th>text1Dormant&lt;sub&gt;eq&lt;/sub&gt;</th>
<th>text1DataEntry&lt;sub&gt;eq&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>text1Dormant&lt;sub&gt;eq&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>text1DataEntry&lt;sub&gt;eq&lt;/sub&gt;</td>
<td>k01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mc-3-6-5 **Events:**

ev-3-6-5-1 **Identifier:** k01
ev-3-6-5-2 **From:** text1DataEntry<sub>eq</sub>
ev-3-6-5-3
To: text1Dormant

Events:
@T_{t0}(keyStroke)

ev-3-6-5-4

vbb-3-7
Modes

md-3-7-1-1 Identifier: text1DataEntry

Description:
Data entry into text1_vbb.

Values:
text1.visualDisplayArea_{eq} : text1DataEntry_{sn}
text1.visualText_{eq} : text1.keyedText_{mq}

Condition:
text1.currentMode_{eq} = $dataEntry$

md-3-7-1-2 Identifier: text1DataEntry

Description:
No activity for text1_vbb.

Values:
instance:visualDisplayArea : text1Dormant_{sn}

Condition:
text1.CurrentMode_{eq} ≠$dataEntry$

vbb-3-8
Demand Functions

df-3-7-1-1 Identifier: keyStroke

df-3-7-1-2 Modes: text1DataEntry

Modes: text1Dormant

Request and Display Description:

<table>
<thead>
<tr>
<th>Modes</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>text1DataEntry</td>
<td>@T_{t0}(keyStroke)</td>
</tr>
</tbody>
</table>

Action: text1.currentMode_{eq} :=$dormant$

vbb-3-9
Periodic Functions
No periodic functions.

[More VBBs would go here]
4 Other Monitored Quantities

This section describes the monitored environmental quantities used by the Home Owner Freedom system that are not described in other parts of the software requirements.

mq-1  

hfUserLanguage

mq-1-1  
Identifier: User preferred language\textsubscript{mq}

mq-1-2  
Description: The language the user prefers for the system dialogue.

mq-1-3  
Hardware/Software:

\[ hfUserLanguage\textsubscript{mq} \in \{hfIsoEn\textsubscript{ck}, hfIsoDe\textsubscript{ck}\} \]

mq-2  
g1hfWindowBLX

mq-2-1  
Identifier: Display Window X Coordinate\textsubscript{mq}

mq-2-2  
Description: X coordinate of the top left corner of the display window

mq-2-3  
Hardware/Software:

\[ g1hfWindowBLX\textsubscript{mq} = 5\text{mm} \]

mq-3  
g1hfWindowBLY

mq-3-1  
Identifier: Display Window Y Coordinate\textsubscript{mq}

mq-3-2  
Description: Y coordinate of the top left corner of the display window

mq-3-3  
Hardware/Software:

\[ g1hfWindowBLY\textsubscript{mq} = 5\text{mm} \]

More Monitored Quantities would go here
5 Other Controlled Quantities

This section describes the controlled environmental quantities used by the Home Owner Freedom system.

cq-1  
**Browser display area**

cq-1-1 | **Identifier:**  g1hfBrowserWindow_\(cq\)

cq-1-2 | **Description:**  GUI display window for the HOF system in a Browser.

cq-1-3 | **Hardware:**  
g1hfBrowserWindow_\(cq\) = g1hfPanel_\(cv\)

cq-2  
**GUI Panels**

cq-2-1 | **Identifier:**  g1hfPanel_\(cq\)

cq-2-2 | **Description:**  GUI output panels for HOF

cq-2-3 | **Hardware:**

| g1hfPanel\(_cq\) = | WHEN |
g1hfPanelModeClass\(_cq\) = |
g1hf100e\(_sn\) | $g1hf100e$ |
g1hf101e\(_sn\) | $g1hf101e$ |
g1hf102e\(_sn\) | $g1hf102e$ |
g1hf110e\(_sn\) | $g1hf110e$ |
g1hf111e\(_sn\) | $g1hf111e$ |
g1hf200e\(_sn\) | $g1hf200e$ |
g1hf201e\(_sn\) | $g1hf201e$ |
g1hf300e\(_sn\) | $g1hf300e$ |
g1hf301e\(_sn\) | $g1hf301e$ |
g1hf320e\(_sn\) | $g1hf320e$ |
g1hf321e\(_sn\) | $g1hf321e$ |
g1hf325e\(_sn\) | $g1hf325e$ |
g1hf100g\(_sn\) | $g1hf100g$ |
<table>
<thead>
<tr>
<th>$g_{1hf101g_{sm}}$</th>
<th>$g_{1hf101g}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_{1hf102g_{sm}}$</td>
<td>$g_{1hf102g}$</td>
</tr>
<tr>
<td>$g_{1hf110g_{sm}}$</td>
<td>$g_{1hf110g}$</td>
</tr>
<tr>
<td>$g_{1hf111g_{sm}}$</td>
<td>$g_{1hf111g}$</td>
</tr>
<tr>
<td>$g_{1hf200g_{sm}}$</td>
<td>$g_{1hf200g}$</td>
</tr>
<tr>
<td>$g_{1hf201g_{sm}}$</td>
<td>$g_{1hf201g}$</td>
</tr>
<tr>
<td>$g_{1hf300g_{sm}}$</td>
<td>$g_{1hf300g}$</td>
</tr>
<tr>
<td>$g_{1hf301g_{sm}}$</td>
<td>$g_{1hf301g}$</td>
</tr>
<tr>
<td>$g_{1hf320g_{sm}}$</td>
<td>$g_{1hf320g}$</td>
</tr>
<tr>
<td>$g_{1hf321g_{sm}}$</td>
<td>$g_{1hf321g}$</td>
</tr>
<tr>
<td>$g_{1hf325g_{sm}}$</td>
<td>$g_{1hf325g}$</td>
</tr>
</tbody>
</table>

More Controlled Quantities would go here
6 Mode Classes

This section describes the mode classes in the HOF system. Each panel in the HOF system is a mode class. Within each panel mode class, mode classes for images are found. The HOF system is always in exactly one panel mode class, one image mode class and one mode at any one time.

Identifier: g1hfPanels

Description:
Describes the sequence in which panels appear to the user and the events that cause the user to move from one panel to the next.

Initial Mode: $outsideHF$

Transitions:
WHEN g1hfUserLanguage = $en$

<table>
<thead>
<tr>
<th>To panel → From panel</th>
<th>$outsideHF$</th>
<th>g1hf00e</th>
<th>g1hf01e</th>
<th>g1hf02e</th>
<th>g1hf10e</th>
<th>g1hf11e</th>
<th>g1hf200e</th>
<th>g1hf201e</th>
<th>g1hf300e</th>
<th>g1hf301e</th>
<th>g1hf320e</th>
<th>g1hf321e</th>
<th>g1hf325e</th>
</tr>
</thead>
<tbody>
<tr>
<td>$outsideHF$</td>
<td>LIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf00e</td>
<td>LIN</td>
<td>N01e</td>
<td>N02e</td>
<td>N03e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf01e</td>
<td>S01e</td>
<td>C01e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf02e</td>
<td>S02e</td>
<td>C02e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf10e</td>
<td>N04e</td>
<td>N05e</td>
<td>N06e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf11e</td>
<td>S06e</td>
<td>C06e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf200e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf201e</td>
<td>S04e</td>
<td>N04e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf300e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf301e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf320e</td>
<td>N05e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf321e</td>
<td>C06e</td>
<td>S06e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g1hf325e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Print Date: September 26, 2001 V02-R11 Home Owner Freedom System
mc-1-5

**Events:**

More events 1 to 25 would go here

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Identifier</th>
<th>From</th>
<th>To</th>
<th>Events</th>
</tr>
</thead>
</table>
| ev-26-1  | S07e       | g1hf325e **mcss** | g1hf325e **mcss** | \@T\textsubscript{0}(g1hf325Save\textsubscript{mq} = $\text{push}$$\)$ WHEN(HFCurrentPanelMode\textsubscript{eq} = $g1hf325e$)
| ev-27-1  | C07        | g1hf325e **mcss** | g1hf320e **mcss** | \@T\textsubscript{0}(g1hf325Cancel\textsubscript{mq} = $\text{push}$$\)$ WHEN(HFCurrentPanelMode\textsubscript{eq} = $g1hf325e$)
| ev-28-1  | C07        | g1hf325e **mcss** | g1hf320e **mcss** | \@T\textsubscript{0}(g1hf325Cancel\textsubscript{mq} = $\text{push}$$\)$ WHEN(HFCurrentPanelMode\textsubscript{eq} = $g1hf325e$$ \land$ HFValidationStatus\textsubscript{eq} = $\text{success}$$)

More Mode Transitions would go here
$A_cE$ Home Owners Freedom Insurance System
System Design

1 Table of Contents

2 Introduction
This document describes the design of the Home Owner Freedom software system.

3 Input Variables
This section describes the input variables used by the Home Owner Freedom system.

iv-1

User interface language

iv-1-1 Identifier: $hf_{UserLanguage_{iv}}$

iv-1-2 Description:
User preferred language for the user interface.

iv-1-4 Relation to Real Time (t):
none

iv-1-5 Data Representation:
$hf_{UserLanguage_{iv}} = hf_{UserLanguage_{mq}}$

iv-1-6 Invalid Values:
$hf_{UserLanguage_{iv}} \neq \{\} \_hFIsSoEn_{ck}, hf_{IsDe}_{ck}$

iv-2 Risk Country

iv-2-1 Identifier: $hf_{RiskCountry_{iv}}$
iv-2-2 **Description:**
Country in which the object to be insured under an HOF policy is located.

iv-2-4 **Relation to Real Time \( (t) \):**
none

iv-2-5 **Data Representation:**

<table>
<thead>
<tr>
<th>hfRiskCounty(_m) = g1h325e:RiskCountry(_m)</th>
<th>hFUserLanguage(_m) = hfIsoEnck</th>
</tr>
</thead>
<tbody>
<tr>
<td>hFUserLanguage(_m) = hfIsoDeck</td>
<td></td>
</tr>
</tbody>
</table>

iv-2-6 **Invalid Values:**
riskCountryNotInsurable\(_m\)

iv-3 **Risk address**

iv-3-1 **Identifier:**
hfRiskAddress\(_iv\)

iv-3-2 **Description:**
Address of the object to be insured under an HOF policy.

iv-3-4 **Relation to Real Time \( (t) \):**
none

iv-3-5 **Data Representation:**

<table>
<thead>
<tr>
<th>hfRiskAddress(_iv).country(_iv) = g1h325e:RiskCountry(_m)</th>
<th>hFUserLanguage(_iv) = hfIsoEnck</th>
</tr>
</thead>
<tbody>
<tr>
<td>hfRiskAddress(_iv).text1(_iv) = g1h325e:RiskAddressText1(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).text2(_iv) = g1h325e:RiskAddressText2(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).text3(_iv) = g1h325e:RiskAddressText3(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).city(_iv) = g1h325e:RiskCity(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).region(_iv) = g1h325e:RiskRegion(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).postCode(_iv) = g1h325e:RiskPostCode(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoEnck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).country(_iv) = g1h325d:RiskCountry(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).text1(_iv) = g1h325d:RiskAddressText1(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).text2(_iv) = g1h325d:RiskAddressText2(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).text3(_iv) = g1h325d:RiskAddressText3(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).city(_iv) = g1h325d:RiskCity(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).region(_iv) = g1h325d:RiskRegion(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
<tr>
<td>hfRiskAddress(_iv).postCode(_iv) = g1h325d:RiskPostCode(_m)</td>
<td>hFUserLanguage(_iv) = hfIsoDeck</td>
</tr>
</tbody>
</table>
iv-3-6  Invalid Values:

<table>
<thead>
<tr>
<th>hFRiskAddress.text1 iv ∈ addressCharsDEck</th>
<th>hFRiskCountry iv =</th>
</tr>
</thead>
<tbody>
<tr>
<td>hFRiskAddress iv ∈ addressCharsENck</td>
<td>hFlsoDegck</td>
</tr>
</tbody>
</table>

iv-3  Urban or rural property.

iv-3-1  Identifier:  hFRiskSetting iv

iv-3-2  Description:
Indicates whether property is urban or rural. Urban properties are within recognized city or town limits and are under the jurisdiction of a professional fire department. Rural properties lie outside city or town limits and are within jurisdictions of volunteer fire departments.

iv-3-4  Relation to Real Time (t):
none

iv-3-5  Data Representation:

<table>
<thead>
<tr>
<th>hFRiskSetting iv = g1h325cRiskSetting smq</th>
<th>hFUserLanguage iv =</th>
</tr>
</thead>
<tbody>
<tr>
<td>hFRiskSetting iv = g1h325dRiskSetting smq</td>
<td>hFlsoDEnck</td>
</tr>
</tbody>
</table>

iv-3-6  Invalid Values:

iv-4  Distance to Hydrant.

iv-4-1  Identifier:  hFRiskToHydrant iv

iv-4-2  Description:
For rural properties, the distance to the RiskTo fire hydrant and the unit provided. The data OK indicates the data has usable values independent of the setting.

iv-4-4  Relation to Real Time (t):
none

iv-4-5  Data Representation:

4Not defined as part of the case study.
### Primary Use.

**Identifier:**  hfRiskPrimaryUse<sub>i,v</sub>  

**Description:**  The primary use of the risk object to be insured.

**Relation to Real Time (t):**  
none

**Data Representation:**

<table>
<thead>
<tr>
<th>hfRiskPrimaryUse&lt;sub&gt;i,v&lt;/sub&gt; = g1hf325e:primaryUse&lt;sub&gt;mq&lt;/sub&gt;</th>
<th>hFUserLanguage&lt;sub&gt;iv&lt;/sub&gt; = hFisoEn&lt;sub&gt;ck&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hfRiskPrimaryUse&lt;sub&gt;i,v&lt;/sub&gt; = g1hf325d:primaryUse&lt;sub&gt;mq&lt;/sub&gt;</td>
<td>hFisoDe&lt;sub&gt;ck&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

### On Risk Period

**Identifier:**  hfRiskOnPeriod<sub>i,v</sub>  

**Description:**  The period during which the insurer is on-risk, and must carry the exposure.

**Relation to Real Time (t):**  
none

**Data Representation:**
### Input Variables

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Relation to Real Time (t):</th>
<th>Data Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invalid Values:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-6-6 Sum insured</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-7-1 Identifier:</td>
<td><strong>hfRiskSumInsured</strong>&lt;sub&gt;<em>iv</em>&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-7-2 Description:</td>
<td>Extent of the exposure to loss for this risk.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-7-4 Relation to Real Time (t):</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-7-5 Data Representation:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Invalid Values:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-8 Button actuated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-8-1 Identifier:</td>
<td><strong>g1hfButtonActuated</strong>&lt;sub&gt;<em>iv</em>&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-8-2 Description:</td>
<td>The button in the Home Owner Freedom system that was actuated by a user.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-8-4 Relation to Real Time (t):</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv-8-5 Data Representation:</td>
<td><strong>g1hfButtonActuated</strong>&lt;sub&gt;<em>iv</em>&lt;/sub&gt; = <strong>g1hfButtonActuated</strong>&lt;sub&gt;<em>mq</em>&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
iv-8-6  **Invalid Values:**

[More Input Variables would go here]
4 Output Variables

This section describes the output variables used by the Home Owner Freedom system.

**ov-1**

**User messages.**

**Identifier:** hfMessageToUser<sub>ov</sub>

**Description:** Message for the user about the state if the system.

**Relation to Real Time (t):**

none

**Data Representation:**

<table>
<thead>
<tr>
<th>hfUserLanguage&lt;sub&gt;cr&lt;/sub&gt;</th>
<th>hfUserLanguage&lt;sub&gt;ce&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hfUserMessage.classCode&lt;sub&gt;ov&lt;/sub&gt; = $523$&lt;br&gt;hfUserMessage.message&lt;sub&gt;ov&lt;/sub&gt; = hfUEiMessageEn&lt;sub&gt;ck&lt;/sub&gt;[523]</td>
<td>t&lt;sub&gt;ck&lt;/sub&gt;</td>
</tr>
<tr>
<td>hfUserMessage.classCode&lt;sub&gt;ov&lt;/sub&gt; = $523$&lt;br&gt;hfUserMessage.message&lt;sub&gt;ov&lt;/sub&gt; = hfUEiMessageDe&lt;sub&gt;ck&lt;/sub&gt;[523]</td>
<td>t&lt;sub&gt;ck&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

More messages would go here

**Invalid Values:**

those above and no other

**ov-2**

**Ue detection reference code.**

**Identifier:** hfUeLctnCode<sub>ov</sub>
ov-2-2  **Description:** Code uniquely identifying the location in the Home Owner Freedom code text where a UE is detected.

ov-2-3  **Relation to Real Time (t):**
none

ov-2-4  **Data Representation:**
\( h_fUeLctnCode_{\text{ov}} = \{1, \ldots, 1254\} \)

ov-2-5  **Invalid Values:**
none

More Output Variable would go here
5 Software Design

5.1 Technical Considerations
Since existing technologies will not be adequate to fully supply the needs of the Home Owner Freedom system a new architecture is needed. The new architecture must integrate the exiting hardware and software platforms and add to them where needs dictate.
This alternative uses the existing SNA network and the existing token ring network. The internal network, the intranet, will give all corporate users on the token ring access to the Home Owner Freedom application. This network will then be extended to associate users. Individual users will have restricted access to the application from the Internet. All networks, except SNA, will use the TCP/IP protocol.

Network Characteristics
This section will describe, in general, the network architecture for Alternative 2.
The general network architecture will look as follows:

Network Architecture Diagram
Corporate intranet
The intranet is currently only used for mail services. This provides an opportunity to use the excess capacity available here and add to it in order to provide the network capacity needed for Home Owner Freedom.

Associate intranet
Associate access to the intranet will be through the Internet using VPNS.

Individual Internet
Individual access to the Home Owner Freedom application will be via the Internet through a security gateway.

Role Out Phases for Home Owner Freedom
The system will be rolled out in three phases as follows:

Phase 1
Access to Home Owner Freedom will be given to corporate intranet users. For this phase the application software must be complete, the application server must be installed, and additional network capacity must be available.

**Phase 2**
Access to Home Owner Freedom will be given to associate intranet users. For this phase a new security gateway must be installed that can support the VPNs with associate users. When this is in place the applications can be made available to associate users.

**Phase 3**
Access to Home Owner Freedom will be given to individual Internet users. In order to provide this level of access the security gateway must have a firewall installed that will only allow limited access to the Home Owner Freedom application.

5.2 Software Development Issues
This section describes designed decisions made during the software development, their reasons and the effects on the Home Owner Freedom software system.

5.2.1 Non-Functional Considerations

**Thin Client**
Non-functional consideration nfc-7 states that NCs will be used as well as PCs. Due to the NC’s restricted local resources a thin client software architecture has been selected. This architecture will be browser based and will download the application and the application data needed for a session from the application server. Therefore, a panel will be an HTML Page and VBBs will be Java™ applets functioning from within the HTML.

**JAR Files**
Since the application must be downloaded from the application server its efficient transmission across the network is an important issue. To optimize the transmission of the application JAR technology will be used. An issue to be addressed later in the design is if the application will be split into JAR files, and if so how will it be split, or if the entire application will be put into the same JAR file.

**Thin Data**

As there is very limited local storage available on the NC’s only data needed frequently by the application will be retained locally, the remaining data will be retained on the application server and on the host.

### 5.3 Package Formation

This section will describe the rational used to form packages.

#### 5.3.1 Encapsulating Data

**Treat Data as Local**

As an effect of sdc-3 application data will be spread across the network. The resulting complexity associated with reading and updating data must be hidden from the application programs. Therefore, the data will be encapsulated so that application programs can always behave as if the data were local.

**CICS Data Access**

Much of the data the Home Owner Freedom application will need resides on the host computer. Most of the data can not, however, be accessed directly and must be accessed through CICS due to relationships in the data that reside in the CICS programs. The part of the CICS interface used for data access must be encapsulated as part of sdc-4.

**Server Data Access**

Part of the application data will reside on the application server and be distribute to users from there. The DBMS, not yet determined, must be encapsulated as part of sdc-4. This will make changes to the DBMS a local change.

**Network Access**
Clearly the application will require a great deal of cross network communication as shown in nfc-8. Since an NC can only run one application at a time, the browser, communication with the server must be done through applets. Applets, however, have the disadvantage that they can only receive data from the server from which they were down loaded[3]. As this is likely to change in future the network access must be encapsulated to avoid the delocalisation of enhancements to network access.

Security
Three levels of access to the application must be provided: corporate, associate and individual access. These different access levels must be rolled out with the releases of the software that open the application to a broader audience. Therefore, application security must be encapsulated to hide the complexity of these changes.

5.3.2 UE Handling

Propagation
UEs handling is done at two distinct levels. First, programs are written to handle incidents locally and if this is not possible then to propagate these UEs to the invoking method. HOF application programs acknowledge the detection of an incident by throwing an exception. Exception objects are then caught by the invoking method and either recovery follows or the UE is propagated up the invocation hierarchy. When the UE is first detected it is given a UEToken. When the UE is propagated up the invocation hierarchy this token in also propagated and allows the trail of propagation to be traced in the UE log.

Localise Changes
Since there are generic UE classes described in the lexicon, detecting a new UE that is part of an already existing UE class will always be a local change. If a new class of UE is detected then the change will delocalised to the site where the UE is detected and to the UE package where the new UE Object class must be defined.

5.3.3 Multiple User Languages

New User Languages
The system currently supports two user languages, English and German. Two sets of panels have been developed to support the two languages. This solution was preferred to others because it provides the most flexibility in addressing user as well as technical needs. In the existing system user needs differ only slightly between language groups. These differences restrict themselves to the default first option in a multiple selection or the default to a German post code in G1HF325xx. The language differences where, however, sufficient to require slight differences in screen layout due to variations in the lengths of words.
Where this design decision will show dividends is when non-European languages are included for users. If Chinese, for example, becomes a user language it is expected that user needs for this group will go beyond the obvious differences in screen format. These additional needs that are language specific are easiest addressed by developing a new set of panels.

5.3.4 Use Packages to Implement Modules

Packages

The Java\textsuperscript{TM} Package is a collection of related classes that provides access protection\cite{18}. As is summarised below, access to data and methods is limited to the package for all designations except public. This makes a package a convenient Java\textsuperscript{TM} language feature for implementing a module.

Package Access

Since a package represents the implementation of a module and modules are typically an abstraction from a base machine to a virtual machine, the classes within the package can share a close coupling. This means that within a module a class may inherit behaviour from one another class that is also within the same package. Inside a package, therefore, access designations can be anything but not public.

<table>
<thead>
<tr>
<th>Access Designations in Java\textsuperscript{TM}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specified access</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>private</td>
</tr>
<tr>
<td>protected</td>
</tr>
<tr>
<td>public</td>
</tr>
<tr>
<td>package</td>
</tr>
</tbody>
</table>
class: accessible only within the same class.
subclass: accessible only within a direct descendent.
package: accessible anywhere within the same package.
system: accessible throughout the system.

sdc-14 Inheritance
Inheritance will be used only within the same package and where ever possible inheritance will be restricted to three generations. Also, inheritance between languages, from Java™ to JavaScript, say, should not be done. The consequences of this type of inheritance for maintenance are poorly understood and this makes the use of this type of inheritance unacceptably risky.

More development decisions would go here

5.3.5 Expected Changes
This section describes the changes the software design has taken into consideration.

exc-1 Additional User Languages
The Home Owner Freedom system is expected to grow into new markets with different languages that may use new alphabets. For this reason panels literals are implemented on the panel. This means the panels must be reimplemented for each new language but this is an acceptable cost.

More expected changes would go here

6 Maintenance Decisions
none.
1 Table of Contents

2 Introduction

2.1 Purpose
This document describes the module structure of the $A_{c}E$ Home Owners Freedom Insurance system. Changes to the module structure of this system will necessarily have an effect on this document and, therefore, changes to the module structure will not be considered complete until this document has been changed to reflect the new module structure.

2.2 Prerequisite Knowledge
Readers are assumed to be familiar with the $A_{c}E$ Home Owners Freedom Insurance Software Requirements document which will be referred to as the Software Requirements. The reader is also assumed to have a general understanding of software and a good understanding of the $A_{c}E$ Insurance non-life product lines.

3 Module Structure
ml-1 Behaviour Hiding Modules
The modules described in this section provide the primary business functions for the system.

ml-1-1 Business Processes
ml-1-1-1 Implemented: Module
ml-1-1-2 Identifier: businessProcesses
ml-1-1-3 Secret:
The implementation of the business process
the hardware and software platform or platforms the task in the business process take place on
security for the business process
ml-1-1-4 Purpose:
This module implements the businesses needed to market and create new business, and to maintain existing business for the Home Owner Freedom products.

ml-1-1-1 Policy Quotations
ml-1-1-1-1 Implemented: Package
ml-1-1-1-2 Identifier: quotations
ml-1-1-1-3 Secret:
What must be done to perform a quotation for a given jurisdiction.
ml-1-1-1-4 Purpose:
This module encapsulates the complexity of calculating premiums. The quotation process calculates the policy premium. This is the amount paid by the insured to transfer responsibility for a risk to the insurer for a period of time, the time period where the responsibility for the risk is with the insurer is referred to the insurers on-risk period [35].

ml-1-1-1 Risk Premium
ml-1-1-1-1 Implemented: class
ml-1-1-1-2 Identifier: riskPremium
ml-1-1-1-3 Secret:
What must be done to calculate the risk premium for a given jurisdiction.
ml-1-1-1-4 **Purpose:**
This module encapsulates the complexity of calculating pure risk premiums for given jurisdictions.

ml-1-1-1-2 **Expenses Loading**
ml-1-1-1-2.1 **Implemented:** class
ml-1-1-1-2.2 **Identifier:** expenses
ml-1-1-1-2.3 **Secret:**
What must be done to calculate the cost of writing a policy.

ml-1-1-1-2.4 **Purpose:**
This module encapsulates the complexity of calculating cost of writing new business for various parts of the corporation and under different jurisdictions.

ml-1-1-1-3 **Profit Loading**
ml-1-1-1-3.1 **Implemented:** class
ml-1-1-1-3.2 **Identifier:** profit
ml-1-1-1-3.3 **Secret:**
What is done to calculate the profit on a policy.

ml-1-1-1-3.4 **Purpose:**
This module encapsulates the complexity of determining how much over cost a policy is sold at.

ml-1-1-1-3 **Contingency Loading**
ml-1-1-1-3.1 **Implemented:** class
ml-1-1-1-3.2 **Identifier:** contingency
ml-1-1-1-3.3 **Secret:**
The amount added to the risk premium to cover variability in claims costs.

ml-1-1-1-3.4 **Purpose:**
This module encapsulates the complexity of assessing the risk for contingencies and determining the cost associated with carrying that type of risk.
Interface Hiding Modules

This section describes those modules that must be changed when external hardware or external software systems are changed.

Device Interfaces Module

Implemented: module
Identifier: deviceInterfaces
Secret: Characteristics of peripheral devices that are likely to change.

Purpose:
The Home Owner Freedom system depends on a great deal of peripheral devices any of which can be replaced by a device that can accomplish the same task but may have a different interface and different behaviour. This module’s purpose is to provide the system with interfaces to the peripheral devices that will not change when the device it encapsulates is replaced.

User Interface Package

Implemented: package
Identifier: aceUI

Secret: The encapsulates of the aceUI

Purpose:
The user interface package comprises all modules used for the on-line interaction with users of the system. The purpose of this module is to shield the system from changes in user interface technology.

Standard Ace GUI services

Implemented: package
Identifier: aceGUI

Secret: The implementation of GUI VBBs

Purpose:
This package contains the implementation of all VBBs in the Home Owner Freedom System. The package consists of Java\textsuperscript{TM} Beans, each implementing a VBB, that are used to populate panels.

ml-2.1.2  
**Standard ace 3270 services**

ml-2.1.2.1  
**Implemented:** package

ml-2.1.2.2  
**Identifier:** ace3270

ml-2.1.2.3  
**Secret:**
The implementation of a 3270 VBBs

ml-2.1.2.4  
**Purpose:**
This package contains the service needed by HOF to provide 3270 services.

More interface hiding modules would go here

ml-3  
**Software Decision Hiding Modules**

This section describes those modules that must change when there are changes to design decision based upon mathematical theorems, physical facts or algorithmic efficiencies.

ml-3.1  
**Financial Calculator**

ml-3.1.1  
**Implemented:** package

ml-3.1.2  
**Identifier:** financialCalculator

ml-3.1.3  
**Secret:**
The implementation of financial calculations.

ml-3.1.4  
**Purpose:**
This package contains programs that do standard financial calculations. The packages purpose is to retain control over how financial calculations are done within the Home Owner Freedom system so that compliance to standards imposed by governing bodies can be guaranteed.

ml-3.1-1  
**Net Present Value of a Lump Sum**

ml-3.1-1.1  
**Implemented:** class

ml-3.1-1.2  
**Identifier:** NPVofValue

ml-3.1-1.3  
**Secret:**
The implementation of the NPV algorithm.
ml-3-1-4  **Purpose:**
This module calculates the present value of a single lump sum over a period of time given an interest rate.

ml-3-1-2  **Net Present Value of a Series of Payments**
ml-3-1-2-1  **Implemented:**  class
ml-3-1-2-2  **Identifier:**  NPVofSeries
ml-3-1-2-3  **Secret:**
The implementation of the NPV algorithm.
ml-3-1-2-4  **Purpose:**
This module calculates the present value of a series of payments made over a period of time given an interest rate.

ml-3-1-3  **Net Future Value of a Lump Sum**
ml-3-1-3-1  **Implemented:**  class
ml-3-1-3-2  **Identifier:**  NFVofValue
ml-3-1-3-3  **Secret:**
The algorithm used for calculating the NFV.
ml-3-1-3-4  **Purpose:**
This module calculates the value of a lump sum at the end of a period of time given an interest rate.

ml-3-1-4  **Net Future Value of a Series of Payments**
ml-3-1-4-1  ** Implemented:**  class
ml-3-1-4-2  ** Identifier:**  NFVofSeries
ml-3-1-4-3  ** Secret:**
The algorithm used for calculating the NFV.
ml-3-1-4-4  ** Purpose:**
This module calculates the value a series of payments have at the end of a period given an interest rate.

ml-3-2  **HOF Data Types Package**
ml-3-2-1  ** Implemented:**  package
ml-3-2-2  ** Identifier:**  hofTypes
ml-3-2.3 **Secret:**
This package makes a secret of the implementation of the data types used in Home Owner Freedom.

ml-3-2.4 **Purpose:**
The purpose of this package is to keep control of the precision of values, particularly financial values, to ensure compliance to the standards imposed by governing bodies for precision and audibility.

ml-3-2.1 **Fixed Point Currency**
ml-3-2-1.1 **Implemented:** class
ml-3-2-1.2 **Identifier:** currency
ml-3-2-1.3 **Secret:**
The implementation of the type.
ml-3-2-1.4 **Purpose:**
This module implements financial data types the require fixed point precision and currency information.

ml-3-2.2 **Audited Fixed Point Currency**
ml-3-2-2.1 **Implemented:** class
ml-3-2-2.2 **Identifier:** currencyA
ml-3-2-2.3 **Secret:**
The implementation of the type.
ml-3-2-2.4 **Purpose:**
This module implements the audit trail needed for some data to comply to accounting audit practices. Old values of the critical fields will be retained by this data type to provide an audit trail of values.

ml-3-2.3 **Character Data**
ml-3-2-3.1 **Implemented:** class
ml-3-2-3.2 **Identifier:** char
ml-3-2-3.3 **Secret:**
The implementation of the type.
ml-3-2-3.4 **Purpose:**
This module implements character data.
ml-3-2-3  **Audited Character Data**
ml-3-2-3.1  **Implemented:** class
ml-3-2-3.2  **Identifier:** charA
ml-3-2-3.3  **Secret:**
The implementation of the type.
ml-3-2-3.4  **Purpose:**
This module implements character data with an audit trail.

[More software decision hiding modules would go here]
1 Table of Contents

2 Introduction 1
3 Package Interfaces 1
4 Class Interfaces 2

2 Introduction
This document is broken into sections where each section describes one package or class interface.

3 Package Interfaces

<table>
<thead>
<tr>
<th>ic-1-1</th>
<th>Package: Ace Standard GUI</th>
</tr>
</thead>
</table>
| ic-1-2 Identifier: aceGUI
| ic-1-3 Description: This package provides the standard used for Ace applications. |
| ic-1-4 Syntax |
| ic-1-4.1 Access Programs: |
4 Class Interfaces

ic-1-1

Class: 30 Character Screen Text Box

ic-1-2

Identifier: Text30GUI

ic-1-3

Package: aceGUI

ic-1-4

Description:
This class provides the behavior for a 30 character text input box. Text entered in the box is always retained in this class, it is for the using program to decide when the text should be returned.

ic-1-6

Syntax

ic-1-5.1

Constructor(s):

ic-1-5.2

Destructor(s):

### Access Programs:

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Access</th>
<th>Returned</th>
<th>Parameter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getActionPerformed_pg()</code></td>
<td>package</td>
<td></td>
<td><code>AceEvent_{tp}</code></td>
</tr>
<tr>
<td><code>getData_pg()</code></td>
<td>package</td>
<td></td>
<td><code>String_{tp}</code></td>
</tr>
<tr>
<td><code>putBLX_pg()</code></td>
<td>package</td>
<td></td>
<td><code>int_{tp}</code></td>
</tr>
<tr>
<td><code>putBLY_pg()</code></td>
<td>package</td>
<td></td>
<td><code>int_{tp}</code></td>
</tr>
</tbody>
</table>

### Canonical Traces:

- canonical(T) \(\Leftrightarrow\)  
- (\(T = \_\)) \(\lor\)  
- (\(T = \text{getActionPerformed}_{pg}()\)) \(\lor\)  
- (\(T = \text{getData}_{pg}()\)) \(\lor\)  
- (\(T = \text{putBLX}_{pg}(\text{blx})\)) \(\lor\)  
- (\(T = \text{putBLY}_{pg}(\text{bly})\))

### Equivalences:

- \(T, \text{Text30GUI}_{pg}() \equiv \text{Text30GUI}_{tp}\)
- \(T, \neg \text{Text30GUI}_{pg}() \equiv \text{undefinedObject}_{er}\)
- \(T, \text{getActionPerformed}_{pg}() \equiv \)
  - Condition: `T = _`  
  - Equivalance: `undefinedObject_{er}`  
  - `keyboardEvent`  
  - `\neg keyboardEvent`  
- \(T, \text{getData}_{pg}() \equiv \)
  - Condition: `T = _`  
  - Equivalance: `undefinedObject_{er}`  
  - `keyboardEvent`  
  - `\neg keyboardEvent`  
- \(T, \text{putBLX}_{pg}() \equiv \)
  - Condition: `T = _`  
  - Equivalance: `undefinedObject_{er}`  
  - `\neg (0\text{blx})`  
  - `0\text{blx}`  
- \(T, \text{putBLY}_{pg}() \equiv \)
  - Condition: `T = _`  
  - Equivalance: `undefinedObject_{er}`  
  - `\neg (0\text{bly})`  
  - `0\text{bly}`
ic-1.8

Values:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = -$</td>
<td>$\text{undefined}_{cr}$</td>
</tr>
<tr>
<td>keyboardEvent</td>
<td>$\text{keyboardAction}_{n}$</td>
</tr>
<tr>
<td>$\neg$keyboardEvent</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = -$</td>
<td>$\text{undefined}_{cr}$</td>
</tr>
<tr>
<td>keyboardEvent</td>
<td>$\text{keyedText}_{n}$</td>
</tr>
<tr>
<td>$\neg$keyboardEvent</td>
<td>$\text{blank}_{ck \times 30}$</td>
</tr>
</tbody>
</table>

More class interfaces would go here

ic-482-1

Class: Message To User

Identifier: $\text{MessageToUserGUI}_{t}$

Package: $aceGUI_{pkg}$

Description:
This class provides the behavior to put message text in the message to the user.

Syntax

Constructor(s):

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Access</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{MessageToUserGUI}_{pq}()$</td>
<td>package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{MessageToUserGUI}_{pq}()$</td>
<td>package</td>
<td></td>
<td>$\text{Language}_{t}$</td>
</tr>
</tbody>
</table>

Destructor(s):

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Access</th>
<th>Parameter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$MessageToUserGUI</td>
<td>package</td>
<td></td>
</tr>
</tbody>
</table>

Access Programs:

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Access</th>
<th>Returned</th>
<th>Parameter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{putCode}_{pq}()$</td>
<td>package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{putLanguage}_{pq}()$</td>
<td>package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{putBLX}_{pq}()$</td>
<td>package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{putBLY}_{pq}()$</td>
<td>package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
ic-482-6  **Canonical Traces:**  
canonical(T) ⇔  

| ic-482.6-1 | (T=)  ) \( \lor  
| ic-482.6-2 | (T=putLanguage_{pg}(language)) \( \lor  
| ic-482.6-3 | (T=putBLX_{pg}(blx)) \( \lor  
| ic-482.6-4 | (T=putBLY_{pg}(bly))  

**Equivalences:**  

| ic-482.7-1 | T.MessageToUserGUI_{pg}(()) ≡ MessageToUserGUI_{tp}  
| ic-482.7-2 | T.¬MessageToUserGUI_{pg}(())  

≡ undefinedObject_{er}  
| ic-482.7-3 | T.putLanguage_{pg}()  

| Condition | Equivalance  
| T= _ | undefinedObject_{er}  
| T≠_ | language  

| ic-482.7-4 | T.putBLX_{pg}()  

| Condition | Equivalance  
| T= _ | undefinedObject_{er}  
| ¬(0\(0\)blx) | invalidX_{er}  
| (0\textbackslash{}blx) | blx  

| ic-482.7-5 | T.putBLY_{pg}()  

| Condition | Equivalance  
| T= _ | undefinedObject_{er}  
| ¬(0\textbackslash{}bly) | invalidY_{er}  
| (0\textbackslash{}bly) | bly  

| ic-482-8 | Values:  

| More class interfaces would go here | n.a. |
A_eE Home Owners Freedom Insurance System
Uses Relation Document

1 Table of Contents

2 Introduction 1
3 Uses Relations 1

2 Introduction
This uses relation document provides a graphical description of the uses relations between modules and also within modules between packages.

3 Uses Relations
This provides the uses relations for the modules described in the Module.
Policy Quotations Package

Legend
- package
- class
- Relation Uses

policyQuotation

[RiskPremium]

[ExpensesLoading]

[ContingencyLoading]

financialTypes
**hob Types Package**

- `hobTypes`
  - `CurrencyAudited`
  - `FinancialValueAudited`
  - `Currency`
  - `FinancialValue`
  - `DateTimeStrm`
  - `javaMath`
  - `javaLangua`

**Legend**
- `package`
- `class`

**Relation Uses**

**More Uses Relations would go here**
bp-1 Blueprint 1
bp-1-1 Identifier: validationStatusSetChecked
bp-1-2 Invocation Information
bp-1-2.1 Parameter List:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvChecked</td>
<td>boolean</td>
</tr>
</tbody>
</table>

bp-1-2.2 Returned: void
bp-1-2.3 Call Syntax:
call validationStatusSetChecked()

bp-1-3 Specification:

\[ R_{validationStatusSetChecked}(\cdot) = \begin{array}{c|c}
\text{pvChecked} = \text{true} & \text{pvChecked} \neq T_{ck} \\
\text{dataChecked'} & T_{ck} \\hline
& F_{ck} \\
\wedge \neg C(pvChecked) & & \\
\end{array} \]

bp-1-4 Internal
bp-1-4.1 Declarations:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataChecked</td>
<td>boolean</td>
</tr>
</tbody>
</table>

bp-1-4.2 Diagram:

(ID-1)

Print Date: September 26, 2001  V02-R11  Home Owner Freedom System
\(A_cE\) Home Owners Freedom Insurance System
Design Audit Trail

1 Development Decisions
This section describes the design decisions made during the software development.

1.1 Non-Functional Considerations
Non-Functional Characteristics for the Home Owner Freedom system are as follows.

nfc-1 Corporate Users:
1500 on SNA network with dumb terminals and further 1000 on token Ring using 3270 emulation.

nfc-2 Associate Users:
Zero at first roll out of the system. Growing to 500 with second release, and 1000 at the end of the project.

nfc-3 Individual Users:
Zero at first roll out of the system. Final version will allow individual users, expected hit rate 1000 of per day after final version is installed.

nfc-4 Rollout:
The system will be rolled out in phases over about five years. The technologies that would be available and used in the final roll out are not known at the time this writing but there is a desire to use new technologies to leverage the marketing and sale of the Home Owner Freedom product.

nfc-5 New Technologies:
As part of using available technologies Ace would like to make information about the product available to the Internet. This access will begin with potential customers being able to do self quotes and will go on to allow potential customers purchase the product directly.

nfc-6 Flexibility:
A flexible sales force has always been key to Ace’s success, therefore, leverage technology for market success must stay a goal in the future

**Cost of Ownership:**
The cost of ownership for new hardware should be kept to a minimum. This will be done by using the existing PCs to run Home Owner Freedom and where new hardware platforms are needed Network Computers will be introduced.

### 1.1.1 Technical alternatives
This section describes the technical alternatives considered at the start of the Home Owner Freedom development.

#### 1.1.1.1 Alternative 1, Traditional Technology
At the inception of the Home Owner Freedom Ace primarily used mainframe technologies. Existing production technologies consisted of an SNA network linked to a Wide Area Network. Access to the mainframe applications used CICS as the dialogue management system which was made available to users through dumb terminal connected directly to SNA and 3270 emulation running on divers PC platforms connected to the WAN. Data access is supplied by IMS and DB/2 DBMSs. Application language is primarily COBOL along with JCL for batch control.

#### 1.1.1.2 Positive Characteristics:
Since the new application will rely, in part, on existing data bases, a development using IMS and or DB2 presented no new requirements here. Also, because the Home Owner Freedom system will rely on existing applications for some of its business processes, interface problems could be reduced by staying on the same hardware and software platform.
It is also argued that the organisation should continue to us proven technologies for which technical staff is available in-house without the need for training.

#### 1.1.1.3 Negative Characteristics:
The traditional solution can not provide the technology needed by the Home Owner Freedom applications marketing goals. An SNA network with 3270 applications can not provide access to the system for all potential users via the Internet. Therefore, this alternative has been dropped.

1.1.1.4 Alternative 2, Augmented Technology
Alternative 2 was used for the software development and is therefore documented in the software design.
1 Constants

This section describes constants used throughout this documentation.

ck-1  German Address Characters
ck-1.1  Constant: addressCharsDE
ck-1.2  Description: Allowable characters for German address text.
ck-1.3  Definition:

\[ \text{addressCharsDE}_{ck} \overset{df}{=} \{ \text{alphaUL}_{ck}, \text{alphaDeAddonUL}_{ck}, \text{numeric}_{ck}, \#", "\" \} \]

ck-2  English Address Characters
ck-2.1  Constant: addressCharsEN
ck-2.2  Description: Allowable characters for English address text.
ck-2.3  Definition:

\[ \text{addressCharsEN}_{ck} \overset{df}{=} \{ \text{alphaUL}_{ck}, \text{numeric}_{ck}, \#", "\" \} \]

ck-3  Upper and Lower Case Alpha
ck-3.1  Constant: alphaUL
ck-3.2  Description: Alpha characters, upper and lower case.
ck-3.3  Definition:

\[ \text{alphaUL}_{ck} \overset{df}{=} \{ \text{alphaU}_{ck}, \text{alphaL}_{ck} \} \]
**ck-4**  
**Upper Case Alpha**

**Constant:** alphaU

**Description:**  
Alpha characters, upper case.

**Definition:**

\[
\text{alphaU}_{ck} \overset{df}{=}
\{ 
\text{"A"}, \text{"B"}, \text{"C"}, \text{"D"}, \text{"E"}, \\
\text{"F"}, \text{"G"}, \text{"H"}, \text{"I"}, \text{"J"}, \\
\text{"K"}, \text{"L"}, \text{"M"}, \text{"N"}, \text{"O"}, \\
\text{"P"}, \text{"Q"}, \text{"R"}, \text{"S"}, \text{"T"}, \\
\text{"U"}, \text{"V"}, \text{"W"}, \text{"Y"}, \text{"Z"} 
\} 
\]

**ck-5**  
**Lower Case Alpha**

**Constant:** alphaL

**Description:**  
Alpha characters, lower case.

**Definition:**

\[
\text{alphaL}_{ck} \overset{df}{=}
\{ 
\text{"a"}, \text{"b"}, \text{"c"}, \text{"d"}, \text{"e"}, \\
\text{"f"}, \text{"g"}, \text{"h"}, \text{"i"}, \text{"j"}, \\
\text{"k"}, \text{"l"}, \text{"m"}, \text{"n"}, \text{"o"}, \\
\text{"p"}, \text{"q"}, \text{"r"}, \text{"s"}, \text{"t"}, \\
\text{"u"}, \text{"v"}, \text{"w"}, \text{"y"}, \text{"z"} 
\} 
\]

**ck-6**  
**German Upper and Lower Case Alpha Specials**

**Constant:** alphaDeAddonUL

**Description:**  
Special upper and lower case alpha characters in the German language.

**Definition:**

\[
\text{alphaDeAddonUL}_{ck} \overset{df}{=}
\{ \text{alphaDeAddonUL} \}
\]
ck-7 German Lower Case Alpha Specials
ck-7.1 Constant: alphaDeAddonL
ck-7.2 Description:
Special lower case alpha characters in the German language.
ck-7.3 Definition:
\[ \text{alphaDeAddonL}_{ck} \overset{df}{=} \{ \ae, \oe, \uuml \} \]

ck-8 German Upper Case Alpha Specials
ck-8.1 Constant: alphaDeAddonD
ck-8.2 Description:
Special upper case alpha characters in the German language.
ck-8.3 Definition:
\[ \text{alphaDeAddonD}_{ck} \overset{df}{=} \{ \ddot{A}, \ddot{O}, \ddot{U} \} \]

ck-9 Number of Home Owner Freedom Recognised Countries
ck-9.1 Constant: allHFCountriesCnt
ck-9.2 Description:
The number of all countries referenced in any way in the Home Owner Freedom system.
ck-9.3 Definition:
\[ \text{allHFCountriesCnt}_{ck} \overset{df}{=} 8 \]

ck-10 Home Owner Freedom Recognised Countries, English
ck-10.1 Constant: allHFCountryNamesEn
ck-10.2 Description:
The names of all countries referenced in any way in the Home Owner Freedom system in English language. Notice that these names may not change their array positions.
ck-10.3 Definition:
allHFCountryNamesEn<sub>ck</sub> ≝
allHFCountryNamesEn<sub>1</sub> = “Canada”
allHFCountryNamesEn<sub>2</sub> = “Great Britain”
allHFCountryNamesEn<sub>3</sub> = “Germany”
allHFCountryNamesEn<sub>4</sub> = “Belgium”
allHFCountryNamesEn<sub>5</sub> = “Luxemburg”
allHFCountryNamesEn<sub>6</sub> = “Holland”
allHFCountryNamesEn<sub>7</sub> = “Mexico”
allHFCountryNamesEn<sub>8</sub> = “Italy”

ck-11 Home Owner Freedom Recognised Countries, German

ck-11.1 Constant: allHFCountryNamesDe

ck-11.2 Description:
The names of all countries referenced in any way in the Home Owner Freedom system in German language. Notice that these names may not change their array positions.

ck-11.3 Definition:
allHFCountryNamesDe<sub>ck</sub> ≝
allHFCountryNamesDe<sub>1</sub> = “Kanada”
allHFCountryNamesDe<sub>2</sub> = “Gro Britannienn”
allHFCountryNamesDe<sub>3</sub> = “Deutschland”
allHFCountryNamesDe<sub>4</sub> = “Belgien”
allHFCountryNamesDe<sub>5</sub> = “Luxemburg”
allHFCountryNamesDe<sub>6</sub> = “Holland”
allHFCountryNamesDe<sub>7</sub> = “Mexico”
allHFCountryNamesDe<sub>8</sub> = “Italien”

ck-12 International Country Codes

ck-12.1 Constant: internationalCountryCodes

ck-12.2 Description:
The International country codes for all Home Owner Freedom recognised countries.

ck-12.3 Definition:
internationalCountryCodes<sub>ck</sub> ≝
allHFCountryNamesDe[1] = “CDA”
allHFCountryNamesDe[6] = “HOL”
allHFCountryNamesDe[7] = “MEX”
allHFCountryNamesDe[8] = “ITA”

ck-13  
**HOF Risk Insurable Countries**

ck-13.1  **Constant:** hFRiskInsurableCountries

ck-13.2  **Description:**
The countries in which risks can be insured as part of the Home Owner Freedom product.

ck-13.3  **Definition:**

\[\text{hFRiskInsurableCountries}_{ck} \equiv \{1,3\}\]

ck-14  
**ISO English code**

ck-14.1  **Constant:** hFlsoEn

ck-14.2  **Description:**
ISO standard Language code for English.

ck-14.3  **Definition:**

\[\text{hFlsoEn}_{ck} \equiv \text{“en”}\]

ck-15  
**ISO German code**

ck-15.1  **Constant:** hFlsoDe

ck-15.2  **Description:**
ISO standard Language code for German.

ck-15.3  **Definition:**

\[\text{hFlsoDe}_{ck} \equiv \text{“de”}\]
2 Images

Although images are a type of constant they will be defined in their own section. Images are graphic files like bit maps (BMP format files) or GIFs. An image can contain other images but cannot take input from any source other than the underlying image file.

Icon with Ace Logo

**Identifier:** aceIcon

**Layout:**

**Details:** x

**Coordinates:** y

**Image Description:** none
im-2  HOF heading
im-2-1  Identifier: litHeadingG1HF24EN
im-2-2  Layout:

Home Owners Freedom

im-2-3  Details:
Font: Halvatia
Size: 24 pt
Text colour: black
Background: transparent
Attributes: none

im-2-4  Coordinates:
none
im-2-5  Image Description:
text literal

im-3  Risk Heading
im-3-1  Identifier: litHeadingRisk24EN
im-3-2  Layout:

RISK

im-3-3  Details:
Font: Halvatia
Size: 24 pt
Text colour: black
Background: transparent
Attributes: none

im-3-4  Coordinates:
none
im-3-5  Image Description:
text literal

im-4  Country Text
im-4-1  Identifier: litCountryEN
im-4-2  Layout:
Risk Country

Print Date: September 26, 2001  V02-R11  Home Owner Freedom System
3 Undesired Events

Although UEs, as described in this section, are a type of constant, their importance makes their being defined in a separate section worth while. The UEs describe in this section are in fact classes of UEs.

4 Incidents

Undefined Object

Identifier: undefinedObject

Description:
A reference was made to an Object that has not yet been allocated.

Message:

hfUEiMessageEn_{242} = “An attempt has been made to use an undefined object.”

hfUEiMessageDe_{242} = “Ein Zugriff auf ein undefiniertes Object wurde durchgeführt.”
String not found in List

**Identifier:** StgNotInList

**Description:**
A keyed list of elements was searched through for the occurrence of a string in the list of keys but no matching string was found in the key values.

**Message:**
\[hfUEiMessageEn_{ek}[523]=\text{“The pattern string could not be found in the List”}\]
\[hfUEiMessageDe_{ek}[523]=\text{“Der Suchbegriff wurde in der Liste nicht gefunden”}\]

5 Crashes

**javaScriptError**

**Identifier:** Java Script Error

**Description:**
When JavaScript is not enabled in a Browser but an attempt is made to execute a JavaScript the browser initiates a window with the following error. The Home Owner Freedom system does not anticipate this error and does not attempt recovery.

**Message:**
Error: JavaScript must be enabled to use NetHelp. You can enable JavaScript by selecting Edit — Preferences — Advanced.
6 Functions

This section defines functions used throughout the software documentation.

fn-1.1 Function: validDate

Description:
Discerns between valid and invalid dates in predefined formats.
returns true when valid data
returns false when otherwise

fn-1.4 Signature:
validDate: Char, DateFormat → Boolean

fn-1.5 Definition:
validDate(date, format) \[ \overset{df}{=} \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>true_{ck}</th>
<th>true_{ck} = checkMMDD_{fn}(date, 4, 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>true_{ck}</td>
<td>format \in {YY-MM-DD} \land</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>true_{ck} = checkMMDD_{fn}(date, 4, 7)</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>format \in {YY-MM-DD} \land</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>true_{ck} = checkMMDD_{fn}(date, 5, 5)</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>format \in {DD-MMYY} \land</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>true_{ck} = checkMMDD_{fn}(date, 1, 3)</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>format \in {MM-DDYY} \land</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>true_{ck} = checkMMDD_{fn}(date, 3, 1)</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>format \in {CC-Yymm} \land</td>
</tr>
<tr>
<td></td>
<td>true_{ck}</td>
<td>true_{ck} = checkMMDD_{fn}(date, 5, 7)</td>
</tr>
<tr>
<td></td>
<td>false_{ck}</td>
<td>otherwise</td>
</tr>
</tbody>
</table>

fn-2.1 Function: checkMMDD

Description:
checks a month has the correct number of days,
returns true when correct number of days in the month
returns false otherwise.
fn-2.4  **Signature:**
checkMMDD: Char,int,int → Boolean

fn-2.5  **Definition:**
checkMMDD(date,startMM,startDD) \(\overset{df}{=} \)

<table>
<thead>
<tr>
<th>checkMMDD' (=)</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>true(_{ck})</strong></td>
<td>((\land \text{date}(\text{startMM for } 2) = 1 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 2 \land \text{date}(\text{startDD for } 2) \leq 29) \lor (\text{date}(\text{startMM for } 2) = 3 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 4 \land \text{date}(\text{startDD for } 2) \leq 30) \lor (\text{date}(\text{startMM for } 2) = 5 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 6 \land \text{date}(\text{startDD for } 2) \leq 30) \lor (\text{date}(\text{startMM for } 2) = 7 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 8 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 9 \land \text{date}(\text{startDD for } 2) \leq 30) \lor (\text{date}(\text{startMM for } 2) = 10 \land \text{date}(\text{startDD for } 2) \leq 31) \lor (\text{date}(\text{startMM for } 2) = 11 \land \text{date}(\text{startDD for } 2) \leq 30) \lor (\text{date}(\text{startMM for } 2) = 12 \land \text{date}(\text{startDD for } 2) \leq 31))</td>
</tr>
<tr>
<td><strong>false(_{ck})</strong></td>
<td>otherwise</td>
</tr>
</tbody>
</table>

fn-3.1  **Function:** validTime

fn-3.3  **Description:**

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
Discerns between valid and invalid times in predefined formats.
returns true when valid data
returns false when otherwise

**fn-3.4**

<table>
<thead>
<tr>
<th><strong>Signature:</strong></th>
<th>validTime: Char, TimeFormat → Boolean</th>
</tr>
</thead>
</table>

**fn-3.5**

<table>
<thead>
<tr>
<th><strong>Definition:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>validTime(time, format) (\stackrel{df}{=})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>validTime' =</th>
<th>Condition</th>
</tr>
</thead>
</table>
| \textbf{true}_{ck} | format \in \{HH-MM-SS-FFF}_{ck}, HH-MM-SS}_{ck} \land \\
| \textbf{true}_{ck} | time(1 for 2) \geq 00 \\
| \textbf{true}_{ck} \land | time(1 for 2) < 60 \\
| \textbf{true}_{ck} \land | time(4 for 2) \geq 00 \\
| \textbf{true}_{ck} \land | time(4 for 2) < 60 \\
| \textbf{true}_{ck} \land | time(7 for 2) \geq 00 \\
| \textbf{true}_{ck} \land | time(7 for 2) < 60 \\
| | otherwise |

<table>
<thead>
<tr>
<th>\textbf{false}_{ck}</th>
</tr>
</thead>
</table>

**fn-3.1**

<table>
<thead>
<tr>
<th><strong>Function:</strong></th>
<th>validTime</th>
</tr>
</thead>
</table>

**fn-3.3**

<table>
<thead>
<tr>
<th><strong>Description:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a Boolean value that is not that provided as a parameter.</td>
</tr>
</tbody>
</table>

**fn-3.4**

<table>
<thead>
<tr>
<th><strong>Signature:</strong></th>
<th>validTime: Boolean → Boolean</th>
</tr>
</thead>
</table>

**fn-3.5**

<table>
<thead>
<tr>
<th><strong>Definition:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>validTime(booleanValue) (\stackrel{df}{=})</td>
</tr>
</tbody>
</table>

If booleanValue = \textbf{true}_{ck} then \textbf{false}_{ck} \textbf{false}_{ck} otherwise

**More Functions would go here**
7 Expressions

This section describes expressions used throughout this documentation.

<table>
<thead>
<tr>
<th>Ex-32.1</th>
<th><strong>Identifier:</strong></th>
<th>riskCountryNotInsurable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex-32.1</td>
<td><strong>Description:</strong></td>
<td>The risk country is not in the set of insurable countries.</td>
</tr>
<tr>
<td>Ex-32.2</td>
<td><strong>Definition:</strong></td>
<td>riskCountryNotInsurable ( \equiv ) getCountryIndex(_p)(hfRiskCountry) ( \notin {{}}) RiskInsurableCountries</td>
</tr>
</tbody>
</table>
Maintenance

as a Source of

Documentation
**A_cE Home Owners Freedom Insurance System Production Failure Log**

<table>
<thead>
<tr>
<th>pfl-320</th>
<th><strong>JavaScript Error</strong></th>
</tr>
</thead>
</table>

**pfl-320-1**

<table>
<thead>
<tr>
<th>Status</th>
<th>Complete</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Notes</th>
<th>None</th>
</tr>
</thead>
</table>

**pfl-320-2**

**Problem owner:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Jack Hump</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Contact ph.</th>
<th>905 525 3472</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Alternate ph.</th>
<th>mobile 950 436 7881</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Pager</th>
<th>905 345 6732 Pager No. 3456</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>eMail</th>
<th><a href="mailto:JackH@ACE.SWmaint.ca">JackH@ACE.SWmaint.ca</a></th>
</tr>
</thead>
</table>

**pfl-320-3**

**Detected by Whom:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Shirly Stuts</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>Home Owner Freedom Call Center consultant</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Ph.</th>
<th>905 345-0101 x234</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>eMail</th>
<th><a href="mailto:ShirlyS@ACE.HOF.ca">ShirlyS@ACE.HOF.ca</a></th>
</tr>
</thead>
</table>

**pfl-320-4**

**When Detected:**

<table>
<thead>
<tr>
<th>Detect Date</th>
<th>2000.11.24</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Detect Time</th>
<th>16:00</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Occurred Date</th>
<th>2000.11.25</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Occurred Time</th>
<th>16:00</th>
</tr>
</thead>
</table>

**pfl-320-5**

**Where Detected:**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>PC</th>
</tr>
</thead>
</table>

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
**H/W specs.** Unknown

**Software** Internet Browser

**S/W Version** unknown

**S/W Release** unknown

**Symptoms:**
The following message appears on screen when Help is clicked in Browser:

```
Error: JavaScript must be enabled to use NetHelp. You can enable JavaScript by selecting Edit — Preferences — Advanced.
```

**Actions Taken:**

- **9:20** Phoned Shirly to ensure she could still work. She could not, that part of the HOF Help system she needs requires JavaScript.

- **9:25** Consulted past entries in the Production Problem log and found and found entry 326. This referred to UEC-234 in the Lexicon but also provided information the problem could be the Browser settings.

- **9:20** Phoned Shirly to find out Browser settings. She had “JavaScript enabled” disabled.

**Follow up Activities:**

None.
UEI-512 unexpected on g1hf325e

Status: Open

Notes:
Error Detected when it appears it should not be while adding risk data for an be policy, policy No. HOF-03452EN.

Problem owner:

Name: Jill Troff
Contact ph.: 905 525 7234
Alternate ph.: mobile 950 436 4744
Pager: 905 345 6732 Pager No. 2774
eMail: JackH@ACE.SWmaint.ca

Detected by Whom:

Name: Jon Aggsuks
Position: Home Owner Freedom Call Center consultant
Ph.: 905 345-0101 x234
eMail: ShirlyS@ACE.HOF.ca

When Detected:

Date: 2000.11.25
Time: 08:05

When Occurred:

Date: 2000.11.25
Time: 08:05

Where Detected:

Hardware: PC
H/W specs.: Unknown
Software: HOF on-line GUI system

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
pfl-342-6-4  **S/W Version**:  02
pfl-342-6-5  **S/W Release**:  11

**Symptoms:**
The following message appears on screen when a country is keyed into the “Risk Country” on panel g1hf325e rather than a selection made from the list. This happens during data capture for risk data when adding a new policy.

| International Country Code was no found in allowed HOF Risk Insurable Countries |

**Actions Taken:**

pfl-342-8-1  **08:10**
Phoned Jon to ensure she could still work - he could but could only enter data with the country code.

pfl-342-8-2  **08:15**
Production Problem log showed no similar Problems. If the problem where part of the migration of changes to the class RiskCountrySeln on 2000.11.15 this problem would have been detected sooner.

pfl-342-8-2  **08:25**
Checked the UE detection reference and the message is indeed coming from RiskCountrySeln.

pfl-342-8-3  **08:30**
Checked production migration log and found the migration for changes to class RiskCountrySeln. This showed the change had been migrated on 2000.11.24, CR-236

pfl-342-8-3  **08:40**
Checked CR-236 to find what had changed. Change was to allow user to enter a country as text but not in this release. This release should only have the validation - it should always be true because it checks against the array displayed in the drop down box. Adding text comes latter.

pfl-342-8-3  **08:50**
Contacted requesting user of CR-236 and found they wanted country code in addition to country name.

pfl-342-8-3  **09:00**
Contacted Programmer, in Milwaukee. in section cr-236-5-1 of CR-326 he understood the user to mean country code instead of country name and wrote change to specs accordingly.

pfl-342-8-3  **09:10**
Contacted release management to follow rollback procedure. Change was rolled back

pfl-342-8-3  **09:15**
Change was rolled back, Production normal.

pfl-342-9  **Follow up Activities:**
Recommend modifying Raising CR-236 so that it clearly states the users requirements.

| More Production Problems would go here |
$A_c E$ Home Owners Freedom Insurance System
Change Requests

More Change Requests would go here

**cr-236-1**  
**Change Request**

**Number**  
236

**Description**
Add Risk Country Validation to g1hf325 panels. Currently this data is captured from a drop down box which excludes all countries in which the ace product is not offered. However, due to legislative changes in the E.U. it will become possible for existing policy holders to insure properties not included in the list in some cases. What cases are not yet known. Therefore an incremental change strategy will be used. In this CR the new validation will be added to the system. In CR-239 the drop down box will be replaced with a drop down that allows the user to key county names. At this point the circumstances about what countries can be included in HOF should have been cleared the new logic will be added as well.

**cr-236-3**  
**Approval History:**

**User Approval**  
Janet Atkins

**Position**  
Call Center Manager

**Date**  
2000.10.15

**IS Approval**  
Fanny Farmer

**Position**  
HOF IT Manager

**Date**  
2000.10.15

**cr-236-4**  
**Contact People:**

**User**  
Jane Doe

**Position**  
Call Center Team Leader

**IT**  
Bill Hupton

Print Date: September 26, 2001   V02-R11   Home Owner Freedom System
Position

Programmer

Impact Analysis:

Documentation
The CR will require adding a validation for when the risk country to risk data capture, G1HF325 screens in all inout languages.

Programs
This change will require adding the validation to the RiskCountry class.

On the following pages the change request would show the parts of the original specification effected by the change request but before the change requests modifications where made. The a copy of the new relevant parts of the specifications would be provided to allow a comparison.

More Change Requests would go here
### A_{cE} Home Owners Freedom Insurance System

#### Change Request Log

<table>
<thead>
<tr>
<th>Change Request</th>
<th>Date</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR-99-245</td>
<td>2000.10.10</td>
<td>Moved to Production</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.10.10</td>
<td>reviewed and approved by user committee</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.10.29</td>
<td>changes complete</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.10.30</td>
<td>Unit test Complete</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.10.31</td>
<td>Integration Test Complete</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.11.04</td>
<td>System Test Complete</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.11.05</td>
<td>User Acceptance ok</td>
</tr>
<tr>
<td>CR-00-723</td>
<td>2000.11.05</td>
<td>Put into production Migration pml-341</td>
</tr>
</tbody>
</table>

---

Nota: This only logs the status of change requests. The CRs themselves are described separately.

---

Print Date: September 26, 2001 | V02-R11 | Home Owner Freedom System
Home Owners Freedom Insurance System
Production Migration Log

**Production Migration**

**Identifier** 341

**Status** Migrated

**Status Date** 2000.11.24

**History**

**Planned** 2000.11.15

**Actual** 2000.11.24 20:00

**Reason** Distribution server has a hard disk failure and needed to be repaired. Production Status Meeting of 2000.11.23 decided to halt all migration to non-host systems until this was repaired.

**Test History**

**Integration** Jill Troff 2000.11.11

**System** Ed Turner 2000.11.12

**Acceptance** Trudy Jude 2000.11.12

**Change Details**

**Change Requests Included** CR-00723

**Program Changes**

<table>
<thead>
<tr>
<th>Element</th>
<th>name</th>
<th>Language</th>
<th>Target System</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>RiskCountrySeln</td>
<td>Java</td>
<td>HOF</td>
</tr>
</tbody>
</table>

**Database Changes**

Print Date: September 26, 2001  V02-R11  Home Owner Freedom System
### pml-341-4-4  **Transaction Changes**

<table>
<thead>
<tr>
<th>Element</th>
<th>name</th>
<th>Language</th>
<th>Target System</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### pml-341-4-5  **Jobstream Changes**

<table>
<thead>
<tr>
<th>Element</th>
<th>name</th>
<th>Language</th>
<th>Target System</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### pml-341-4-6  **Other Changes**

<table>
<thead>
<tr>
<th>Element</th>
<th>name</th>
<th>Language</th>
<th>Target System</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### pml-341-5  **Rollout**

### pml-341-5-1  **Procedure**
standard class role out

### pml-341-5-2  **Notes**
None

### pml-341-6  **Rollback**

### pml-341-6-1  **Procedure**
Redistribute previous version of class. Versions are compatible.

### pml-341-6-2  **Notes**
None

---

More Production Migrations would go here
## Ace HOF Index

| ace3270       | AT-, 1, 1
|              | SD-, 3
| as Module     | addressCharsEN  
| MG-, 5        | as Constant 
| AceEvent      | AT-, 1, 1
| as Type       | SD-, 3
| MG-, 3        | addressCityDataEntry
| aceGUI        | as Mode 
| as Module     | SR-, 18, 19, 21, 23
| MG-, 5        | addressCountry
| as Package    | as Controlled Quantities
| MG-, 2        | SR-, 16
| MF-, 4        | addressCurrentMode
| as Type       | SR-, 18–23
| MG-, 1        | addressDormant
| AceIcon       | as Mode 
| as Image      | SR-, 18, 22
| ST-, 13       | addressPostCodeDataEntry
| aceIcon       | as Mode 
| as Image      | SR-, 18, 22, 23
| LX-, 6        | addressPostCodeText
| SR-, 4        | as Mode 
| aceUI         | SR-, 22
| as Module     | addressRegionDataEntry
| MG-, 4        | as Mode 
| actuationState| SR-, 18, 21, 23
| as Controlled Quantities | addressRegionText
| SR-, 8        | as Mode 
| address       | SR-, 21
| as Image      | addressT1DataEntry
| SR-, 19–22    | as Mode 
| as Input Variables | addressCharsDE
| -, 169        | as Constant
| SR-, 3, 14, 22|
SR-, 18, 20, 23
addressT2DataEntry
as Mode
SR-, 18, 20, 23
addressT3DataEntry
as Mode
SR-, 18, 20, 23
addressTab
as Demand Operation
SR-, 23
addressText
as Monitored Quantities
SR-, 16, 16, 19, 20
allHFCountriesCnt
as Constant
AT-, 3, 3
allHFCountryNamesDe
as Constant
AT-, 4, 4
allHFCountryNamesEn
as Constant
AT-, 3, 4
alphaDeAddonD
as Constant
AT-, 3, 3
alphaDeAddonL
as Constant
AT-, 2, 3
alphaDeAddonUL
as Constant
AT-, 1, 2, 2
alphaL
as Constant
AT-, 1, 2, 2
alphaU
as Constant
AT-, 1, 1, 2
alphaUL
as Constant
AT-, 1, 1, 1
blank
as Constant
MI-, 4
SR-, 7
BLX
as Controlled Quantities
SR-, 3, 4, 4, 5, 15, 16, 25, 26,
26, 27, 32, 32, 33, 33
ST-, 10, 11
BLY
as Controlled Quantities
SR-, 3, 5, 5, 15, 16, 25, 26,
26, 27, 32, 33, 33
ST-, 10, 11
businessProcesses
as Module
MG-, 2
C07
as Event
SR-, 40
CCYYMMDD
as Constant
LX-, 10
cdnProvinces
as Constant
SR-, 16
char
as Module
MG-, 7
charA
as Module
MG-, 8
check
as Monitored Quantities
SR-, 6
checkBoxVirtical as Input Variables SR-, 3
checkMMDD as Function LX-, 10, 11
choice as Controlled Quantities SR-, 7
city as Monitored Quantities SR-, 6, 16, 17
Class, 26
contingency as Module MG-, 3
countrySel as Input Variables SR-, 3
currency as Controlled Quantities SR-, 8 as Module MG-, 7
currencyA as Module MG-, 7
CurrentMode as Controlled Quantities SR-, 34, 34, 35
currentMode as Controlled Quantities SR-, 29, 35
date as Controlled Quantities SR-, 7, 8
dateTime as Input Variables SR-, 4 DDMMYY as Constant LX-, 10
deuStates as Constant SR-, 16
deviceInterfaces as Module MG-, 4 Display Window X Coordinate as Monitored Quantities SR-, 36, 36 Display Window Y Coordinate as Monitored Quantities SR-, 36, 36 displayArea as Controlled Quantities SR-, 5, 16, 27, 33
distance as Controlled Quantities SR-, 7 distanceSel as Input Variables SR-, 3 End Point, 26 enteredCode as Monitored Quantities SR-, 17 Entity, 26 exitButton as Input Variables SR-, 4 as Monitored Quantities SR-, 8, 8
expenses
  as Module
    MG-, 3

F
  as Constant
    UR-, 1
false
  as Constant
    LX-, 10–12
financialCalculator
  as Module
    MG-, 5
glhfl00e
  as Image
    SR-, 37
glhfl100g
  as Image
    SR-, 37
glhfl101e
  as Image
    SR-, 37
glhfl101g
  as Image
    SR-, 38
glhfl102e
  as Image
    SR-, 37
glhfl102g
  as Image
    SR-, 38
glhfl110e
  as Image
    SR-, 37
glhfl110g
  as Image
    SR-, 38
glhfl111e
  as Image
    SR-, 37
glhfl111g
  as Image
    SR-, 38
glhf200e
  as Image
    SR-, 37
glhf200g
  as Image
    SR-, 38
glhf201e
  as Image
    SR-, 37
glhf201g
  as Image
    SR-, 38
glhf300e
  as Image
    SR-, 37
glhf300g
  as Image
    SR-, 39
as Image
SR-, 38

ghf301e
as Image
SR-, 37
as Mode Class
SR-, 39

ghf301g
as Image
SR-, 38

ghf320e
as Image
SR-, 37
as Mode Class
SR-, 39, 40

ghf320g
as Image
SR-, 38

ghf321e
as Image
SR-, 37
as Mode Class
SR-, 39

ghf321g
as Image
SR-, 38

ghf325Cancel
as Monitored Quantities
SR-, 40

G1HF325E
as Panel
SR-, 4

ghf325e
as Image
SR-, 37
as Mode Class
SR-, 9, 39, 40
as Panel
SR-, 169
SR-, 5

ghf325e riskCountry
as Monitored Quantities
SR-, 6

ghf325eAddressDtEn
as Mode
SR-, 9

ghf325eaddressDtEn
as Mode
SR-, 9, 10, 12

ghf325eBtnExit
as Mode
SR-, 9

ghf325eBtnExitFsOn
as Mode
SR-, 11

ghf325eBtnExitFsOnly
as Mode
SR-, 12

ghf325eBtnNext
as Mode
SR-, 9, 11, 12

ghf325eBtnNextFsOn
as Mode
SR-, 11

ghf325eBtnNextFsOnly
as Mode
SR-, 12

ghf325eBtnSave
as Mode
SR-, 9, 11, 12

ghf325eBtnSaveFsOn
as Mode
SR-, 11

ghf325eBtnSaveFsOnly
as Mode
SR-, 12
g1hf325eCountryDtEn
  as Mode
  SR-, 9, 11, 12

SR-, 40

g1hf325eCurrentMode
  as Controlled Quantities
  SR-, 9-12

SR-, 37, 38

g1hf325eEndDtEn
  as Mode
  SR-, 9, 10, 12

SR-, 12

g1hf325eFromDx
  as Mode
  SR-, 10

SD-, 5

g1hf325eNearestHydrantDtEn
  as Mode
  SR-, 9, 10, 12

SD-, 5

g1hf325ePrimaryUseDtEn
  as Mode
  SR-, 9, 12


g1hf325ePrimaryUseDtEn
  as Mode
  SR-, 10


g1hf325ePanel
  as Mode Class
  SR-, 37


g1hf325ePrimaryUseDtEn
  as Mode
  SR-, 10


g1hfPanel
  as Controlled Quantities
  SR-, 37


g1hfPanelModeClass
  as Controlled Quantities
  SR-, 37


g1hfPanels
  as Mode Class
  SR-, 39


g1hfUserLanguage
  as Input Variables
  SR-, 39


g1hfWindowBlx
  as Monitored Quantities
  SR-, 36


g1hfWindowBLX
  as Monitored Quantities
  SR-, 36


gbrShires
  as Constant
  SR-, 16


gbrShires
  as Constant
  SR-, 16


gbrShires
  as Constant
  SR-, 16


gbrShires
  as Constant
  SR-, 16


cGetCountryIndex
  as Function
  LX-, 13


Print Date: September 26, 2001
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Constants</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFCurrentPanelMode</td>
<td>as Controlled Quantities</td>
<td>SR-, 40</td>
<td></td>
</tr>
<tr>
<td>hFSelDe</td>
<td>as Constant</td>
<td>AT-, 5, 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SD-, 2-5, 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR-, 1, 36</td>
<td></td>
</tr>
<tr>
<td>hFIsioEn</td>
<td>as Constant</td>
<td>AT-, 5, 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SD-, 2-5, 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR-, 1, 36</td>
<td></td>
</tr>
<tr>
<td>hfMessageToUser</td>
<td>as Output Variables</td>
<td>SD-, 7, 7</td>
<td></td>
</tr>
<tr>
<td>hfMessageToUser.text</td>
<td>as Output Variables</td>
<td>SR-, 5</td>
<td></td>
</tr>
<tr>
<td>hfMessageToUser.uEclassCode</td>
<td>as Output Variables</td>
<td>SR-, 5</td>
<td></td>
</tr>
<tr>
<td>hFRiskAddress</td>
<td>as Input Variables</td>
<td>SD-, 3</td>
<td></td>
</tr>
<tr>
<td>hfrRiskAddress</td>
<td>as Input Variables</td>
<td>SD-, 2, 3</td>
<td></td>
</tr>
<tr>
<td>hfrRiskAddress.city</td>
<td>as Input Variables</td>
<td>SD-, 2</td>
<td></td>
</tr>
<tr>
<td>hfrRiskAddress.country</td>
<td>as Input Variables</td>
<td>SD-, 2</td>
<td></td>
</tr>
<tr>
<td>hfrRiskAddress.postCode</td>
<td>as Input Variables</td>
<td>SD-, 2</td>
<td></td>
</tr>
<tr>
<td>hfrRiskAddress.region</td>
<td>as Input Variables</td>
<td>SD-, 2</td>
<td></td>
</tr>
</tbody>
</table>
SD-, 5
hfriskOnStartTme
as Input Variables
SD-, 5
hfriskPrimaryUse
as Input Variables
SD-, 4, 4
hfriskSetting
as Input Variables
SD-, 3, 3
hfriskSumInsured
as Input Variables
SD-, 5, 5
hfriskSumInsuredCurrency
as Input Variables
SD-, 5
hfriskSumInsuredValue
as Input Variables
SD-, 5
hfriskToHydrant
as Input Variables
SD-, 3, 4
hfriskToHydrantDataOK
as Input Variables
SD-, 4
hfriskToHydrantDistance
as Input Variables
SD-, 4
hfriskToHydrantDistanceUnits
as Input Variables
SD-, 4
hfueiMessageDe
as Constant
LX-, 8, 9
SD-, 7
hfueiMessageEn
as Constant
LX-, 8, 9
SD-, 7
hflctncode
as Output Variables
SD-, 7, 8, 8
SR-, 5
hfUserLanguage
as Input Variables
SD-, 2-5
hfUserLanguage
as Input Variables
SD-, 7
SR-, 1, 1
as Monitored Quantities
SR-, 1, 36
hfUserMessage.classCode
as Output Variables
SD-, 7
hfUserMessage.message
as Output Variables
SD-, 7
HFValidationStatus
as Controlled Quantities
SR-, 40
HH-MM-SS
as Constant
LX-, 12
HH-MM-SS-FFF
as Constant
LX-, 12
HHMMSS
as Constant
LX-, 12
HHMMSSFF
as Constant
LX-, 12
hofTypes
as Module
MG-, 7
Individual User, 28
   inputText
      as Monitored Quantities
         SR-, 27, 28, 28, 29, 30
   int
      as Type
         MG-, 3
         MI-, 4
   internationalCountryCodes
      as Constant
         AT-, 4, 4
internet, 27
intranet, 27

Java Archive, 27
Java Script Error
   as UE Crash
      LX-, 9

K01
   as Event
      SR-, 29
k01
   as Event
      SR-, 35
keyboardAction
   as Function
      MI-, 4
KeyedText
   as Monitored Quantities
      SR-, 16, 19–22
keyedText
   as Controlled Quantities
      SR-, 34
   as Function
      MI-, 4
   as Monitored Quantities
      SR-, 27–30, 33, 34, 34, 35
KeyStroke
   as Monitored Quantities
      SR-, 34
keyStroke
   as Demand Operation
      SR-, 35
Language
   as Type
      MI-, 4
litCityEN
   as Image
      SR-, 4
litCountryEN
   as Image
      LX-, 8
      SR-, 4
litEndDateDateTimeEN
   as Image
      SR-, 4
literalRiskAddressDE
   as Image
      SR-, 4, 6
litHeadingG1HF24EN
   as Image
      LX-, 7
litHeadingG1HF24EN
   as Image
      SR-, 4
litHeadingRisk24EN
   as Image
      LX-, 7
      SR-, 4
litHOFEN
   as Image
      SR-, 4
litHydrantDistEN
   as Image
SR-, 4
litOnRiskPeriodEN
   as Image
   SR-, 4
litPrimaryUseEN
   as Image
   SR-, 4
litRegionEn
   as Image
   SR-, 4
litRiskAddressEN
   as Image
   SR-, 4
litRuralEN
   as Image
   SR-, 4
litStartDateTimeEN
   as Image
   SR-, 4
litSumInsuredEN
   as Image
   SR-, 4
litUrbanEN
   as Image
   SR-, 4

messageToUser
   as Controlled Quantities
   SR-, 5, 6
   as Input Variables
   SR-, 4
messageToUser.text
   as Controlled Quantities
   SR-, 5
messageToUser.uEclassCode
   as Controlled Quantities
   SR-, 5
messageToUser.uELctnCode
   as Controlled Quantities
   SR-, 5

as Controlled Quantities
   SR-, 5
MessageToUserGUI
   as Type
   MI-, 4, 5
Method, 27
MMDDYY
   as Constant
   LX-, 10
Module, 27

nearestHydrant
   as Monitored Quantities
   SR-, 6, 7
nearestHydrantDataOk
   as Monitored Quantities
   SR-, 7
nearestHydrantDistance
   as Monitored Quantities
   SR-, 7
nearestHydrantUnits
   as Monitored Quantities
   SR-, 7
Network Computer, 27

nextButton
   as Input Variables
   SR-, 4
   as Monitored Quantities
   SR-, 8, 8
NFVofSeries
   as Module
   MG-, 6
NFVofValue
   as Module
   MG-, 6
NPVofSeries
   as Module
   MG-, 6
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPVoFValue</td>
<td>numeric as Module MG-, 6</td>
</tr>
<tr>
<td>onRiskEnd</td>
<td>as Monitored Quantities SR-, 7, 8</td>
</tr>
<tr>
<td>onRiskEndDate</td>
<td>as Monitored Quantities SD-, 5 SR-, 8</td>
</tr>
<tr>
<td>onRiskEndTime</td>
<td>as Monitored Quantities SD-, 5 SR-, 8</td>
</tr>
<tr>
<td>onRiskStartTime</td>
<td>as Monitored Quantities SD-, 5 SR-, 7</td>
</tr>
<tr>
<td>Package</td>
<td>27</td>
</tr>
<tr>
<td>Personal Computer</td>
<td>27</td>
</tr>
<tr>
<td>postCode</td>
<td>as Input Variables SR-, 15 ST-, 13 as Monitored Quantities SR-, 6, 17, 17</td>
</tr>
<tr>
<td>postCodeCurrentMode</td>
<td>as Controlled Quantities SR-, 19–22</td>
</tr>
<tr>
<td>primaryUse</td>
<td>as Input Variables SR-, 4 as Monitored Quantities SD-, 4 SR-, 7, 7</td>
</tr>
<tr>
<td>profit</td>
<td>as Module MG-, 3</td>
</tr>
<tr>
<td>push</td>
<td>as Controlled Quantities SR-, 8</td>
</tr>
<tr>
<td>pushButton</td>
<td>as Demand Operation SR-, 12, 13</td>
</tr>
<tr>
<td>quotations</td>
<td>as Module MG-, 2</td>
</tr>
<tr>
<td>region</td>
<td>as Monitored Quantities SR-, 6, 17, 17</td>
</tr>
<tr>
<td>regionSel</td>
<td>as Controlled Quantities SR-, 16, 16</td>
</tr>
<tr>
<td>Restart</td>
<td>28</td>
</tr>
<tr>
<td>riskAddress</td>
<td>as Input Variables ST-, 13 as Monitored Quantities SR-, 6, 6</td>
</tr>
<tr>
<td>RiskAddressCity</td>
<td>as Monitored Quantities SR-, 6</td>
</tr>
</tbody>
</table>
RiskAddressPostCode
  as Monitored Quantities
    SD-, 6
  RiskAddressRegion
    as Monitored Quantities
      SR-, 6
RiskAddressText1
  as Monitored Quantities
    SD-, 2
    SR-, 6
RiskAddressText2
  as Monitored Quantities
    SD-, 2
    SR-, 6
RiskAddressText3
  as Monitored Quantities
    SD-, 2
    SR-, 6
RiskCity
  as Monitored Quantities
    SD-, 2
RiskCountry
  as Monitored Quantities
    SD-, 2
riskCountry
  as Monitored Quantities
    SR-, 6, 6
riskCountryNotInsurable
  as Expression
    LX-, 13
    SD-, 2, 7
RiskPostCode
  as Monitored Quantities
    SD-, 2
riskPremium
  as Module
    MG-, 3
RiskRegion
SR-, 6, 6, 7
Start Point, 28
state
   as State
   ST-, 17
StgNotInList
   as UE Incident
   LX-, 9
String
   as Type
   MG-, 3
sumInsured
   as Input Variables
   SR-, 4
   as Monitored Quantities
   SR-, 8, 8
sumInsuredCurrency
   as Monitored Quantities
   SD-, 5
   SR-, 8
sumInsuredValue
   as Monitored Quantities
   SD-, 5
   SR-, 8
T
   as Constant
   UR-, 1
t
   as Constant
   SD-, 7
T01
   as Event
   SR-, 9, 18
T02
   as Event
   SR-, 10, 18
T03
   as Event
   SR-, 10, 18
   T04
   as Event
   SR-, 10, 19
   T05
   as Event
   SR-, 10, 19
   T06
   as Event
   SR-, 10, 19
   T07
   as Event
   SR-, 10
   T08
   as Event
   SR-, 11
   T09
   as Event
   SR-, 11
   T10
   as Event
   SR-, 11
   T11
   as Event
   SR-, 11
   T30
   as Event
   SR-, 29
test1
   as Input Variables
   -, 169
text
   as Monitored Quantities
   SR-, 6, 16
text1
   as Input Variables
   -, 169
SR-, 25, 26, 29, 30, 32, 35
ST-, 11, 20
as Mode Class
SR-, 34
text15
as Input Variables
SR-, 15
text15CurrentMode
as Controlled Quantities
SR-, 19–22
text15CurrentMode
as Controlled Quantities
SR-, 27, 27, 31
text15DataEntry
as Image
SR-, 33, 35
as Mode
SR-, 22, 30, 34, 35, 35
text15Dormant
as Image
SR-, 33, 35
as Mode
SR-, 31, 34, 35, 35
Text30
as Input Variables
SR-, 24
text30
as Input Variables
-, 169
SR-, 15, 19–21
as Mode Class
SR-, 18, 29
text30:BLX
as Controlled Quantities
SR-, 14, 24
text30:BLY
as Controlled Quantities
SR-, 14, 25
text30CurrentMode
as Controlled Quantities
SR-, 19–22, 29
text30DataEntry
as Image
SR-, 16, 27, 29
as Mode
SR-, 29
text30Dormant
as Image
SR-, 16, 27, 31
as Mode
SR-, 29
Text30GUI
as Type
MG-, 2, 3
time
as Controlled Quantities
SR-, 7, 8
ture
as Constant
LX-, 10–12
TRX
as Controlled Quantities
SR-, 5, 5, 15, 15, 26, 26, 27, 33
TRY
as Controlled Quantities
SR-, 5, 5, 15, 16, 26, 26, 27, 33
undefinedObject
as UE Incident
LX-, 8
unit
as Controlled Quantities
SR-, 7
User prefered language
as Monitored Quantities
SR-, 36, 36
validDate
  as Function
  LX-, 10
validTime
  as Function
  LX-, 12
value
  as Controlled Quantities
  SR-, 8
Virtual Private Network, 28
visualDisplayArea
  as Controlled Quantities
  SR-, 5, 6, 16, 19–22, 26, 27,
  29, 31, 33, 33, 35
visualText
  as Controlled Quantities
  SR-, 27, 27, 33, 33, 35

World Wide Web, 28
x
  as Mode
  ST-, 17
  as Mode Class
  ST-, 17
YY-MM-DD
  as Constant
  LX-, 10
YYMMDD
  as Constant
  LX-, 10
zero
  as Constant
  SR-, 7