National Physical Laboratory

Report

DEM-ES 013

Software Support for Metrology Best Practice Guide No. 7

Development and Testing of Spreadsheet Applications

R M Barker, P M Harris and G I Parkin

Revised: December 2006

dti
FUNDED BY THE DTI

Software Support for Metrology
Best Practice Guide No 7

Development and Testing of Spreadsheet Applications

R M Barker, P M Harris and G I Parkin
Mathematics and Scientific Computing Group

December 2006

ABSTRACT

This guide gives best practice on the use of applications implemented as spreadsheets, so that they may be properly validated. It takes the approach that development and testing of spreadsheet applications should be founded on proper software engineering principles, bearing in mind that spreadsheets have aspects that are not found in normal software development and testing.

ISSN 1744–0475

National Physical Laboratory
Hampton Road, Teddington, Middlesex, United Kingdom. TW11 0LW

We gratefully acknowledge the financial support of the UK Department
of Trade and Industry (National Measurement System Directorate)

Approved on behalf of the Managing Director, NPL by
Jonathan Williams, Knowledge Leader of the Electrical and Software team

Table of Contents

1

Introduction to Spreadsheet Applications

IN THIS CHAPTER

1. Introduction

1.1 Overview

This guide was produced as part of the SS*f*M project "Testing spreadsheets and other packages used in metrology". The aim of this project is to contribute to the development of an infrastructure, comprising supporting information and guidelines, to ensure that the use of software, particularly spreadsheets, within metrology is made as effective as possible. This is to be achieved in two ways: firstly, by investigating the numerical accuracy of the intrinsic and in-built functions provided by spreadsheets; and secondly, by giving best practice on the development of spreadsheets, including how to enable users to test their spreadsheet implementations against a design. This guide addresses the latter issue; the issue of testing numerical accuracy is addressed in earlier reports from the *SSf*M programme [1-3] (also see [4-6] for further work relating to numerical testing of spreadsheets). It is hoped that by addressing the two aspects, numerical accuracy and software design for testability, the quality of the measurement results delivered by spreadsheets used in the day-to-day operations of measurement and calibration laboratories can be assured.

To enable a spreadsheet application to be tested, it must have been developed properly; so that testing and (where appropriate) validation can become part of the development process. This leads to the main slogan of this guide:

"creating a spreadsheet is software development and should follow sound software engineering principles."

The needs of testing must be addressed during the design and development of a spreadsheet application and we aim to give guidance on the design and development of spreadsheet applications with testing in mind – so the second slogan is ***"design for testability"***. The complexity of the underlying spreadsheet package precludes very rigorous validation of spreadsheet applications and so spreadsheets should not be used for high integrity applications.

The guide pays attention to the way in which spreadsheet applications are often developed, starting with some typical data and seeing, by trial and error, what calculations are sensible and what layouts of input and output are appealing. This trial and error approach to implementation has been legitimised by the name "rapid prototyping" by software engineers [7]. This approach gives the application writer a great deal of understanding of the nature of the problem to be solved by the application: from requirements, through specification and design, to testing and validation. One of the aims of the guide is to put "rapid prototyping" spreadsheet application development within the framework of software development.

This guide is not aimed at any particular existing spreadsheet package but aims to give practical advice in terms of existing spreadsheet technology, while allowing for improvements in the technology; indeed some of the suggestions may not be possible in existing packages. The facilities provided by spreadsheet packages are increasing all the time: a spreadsheet application developer should be willing to use whatever technology in spreadsheet packages is introduced, if this technology will improve the ease, with which the application is developed, or the usability, testability and maintainability of the application.

The next section describes the terms used (in this guide and elsewhere) in discussing spreadsheets. The final section of the introduction gives our thoughts (as software engineers) on how we approached the development of the guide. Chapter two describes the central issues for a software engineering approach to spreadsheet applications: risk analysis, software development lifecycle and the testing of spreadsheet applications. Chapter three describes techniques that can be used to answer particular issues raised in chapter two, including a radical technique to migrate the application away from using spreadsheets. Chapter four gives checklists that can be used to determine if the requirements have been met in particular aspects of spreadsheet application development and testing, with reference to the techniques in chapter three.

Appendix A is a template for procedures for the production of spreadsheet applications following the guide, with references to chapters two, three and four. Appendix B describes the NPL test harness for Excel – EUnit. Appendix C gives references to documents that were used in developing the guide and documents that may provide further useful guidance.

Acknowledgements

This guide was produced under the Software Support for Metrology programme, which is managed on behalf of the DTI by the National Physical Laboratory. For more information on the Software Support for Metrology programme visit the SS*f*M website at www.npl.co.uk/ssfm or, contact the programme manager by phone (on 020 8943 7100), email (ssfm@npl.co.uk) or by post to National Physical Laboratory, Teddington, Middlesex, UK TW11 0LW.

Practical examples of procedures for the design, development and testing of spreadsheets were provided by Ted Scorer (then of Glaxo-Wellcome), and we gratefully acknowledge his help in this work. The guide was originally produced in 2000 and revised in 2004, following comments from staff in NPL's Centre for Mathematics and Scientific Computing, in particular Edward Johnson.

This revision was produced in 2006 with a changed layout. The revision to the guide followed comments from the accompanying course "Developing Spreadsheet Applications for Measurement" held in November 2005 and November 2006: we thank the participants for their input.

1.2 Glossary

NOTE

Confusing terms for spreadsheets

These boxes are used to highlight difficult and important sections. Many terms are used in connection with spreadsheet applications, the same terms (e.g. "spreadsheet" itself) for different concepts, and many different terms for the same concepts.

It is necessary to define some terms that will be used throughout the guide. Some terms (e.g. "spreadsheet") can be used with different meaning depending on the context: to avoid this confusion, the terms below will be used consistently. The glossary also defines some terms that are also used to refer to spreadsheet concepts but which, as stated below, are not used in this guide; the terms are included here to show how they relate to terms used in the guide. The qualifying words {in brackets} may be omitted if they are clear from the context.

{spreadsheet} application	software developed by a user of a spreadsheet package, consisting of a program and input data
{spreadsheet} code	both (1) formulae in cells, and (2) the definitions of functions and procedures
{spreadsheet application} developer	a developer of a spreadsheet application
{spreadsheet} end-user	an alternative term for "operator" (not used in this guide)
data	numbers and text that appear in a spreadsheet
function	a user defined operation that returns a value to be used in a cell formula
input data	data entered into cells by the operator
intermediate data	data produced by formulae as an intermediate step in the data processing
macro	a user-defined operation which can be called to manipulates the spreadsheet directly (rather than returning a value in a formula) – a procedure without parameters
macro language	the programming language that is used to define user-defined operations, including macros and user-defined functions
{spreadsheet} model	a computer model implemented as a spreadsheet application (not used in this guide)

operator	a simple user of a spreadsheet application: enters data but cannot change the application (see “end-user” above)
output data	data produced by formulae as the final step of data processing
{spreadsheet} package	software for creating and developing spreadsheets
procedure	a user-defined operation that does not return a value
{spreadsheet application} program	worksheets with program data, formulae, functions and procedures.
program data	data entered into cells by the developer: constants used by the program, which are not to be changed by the operator
spreadsheet	a spreadsheet package in use by a user, with worksheets containing data, formulae, macros, functions, etc.
{spreadsheet} tool or ***{spreadsheet} toolkit***	a spreadsheet package (not used in this guide)
{spreadsheet} user	someone who uses a spreadsheet package to process data, by entering data and/or adding/amending formulae/code; thus, they may take on the roles of operator and developer.
user-defined function	a function that can be called by formula in a cell
worksheet	an individual page/sheet/table of a spreadsheet

Some confusing terms:

- spreadsheet

  A spreadsheet can mean a spreadsheet package (such as Excel, Lotus, etc.), a collection of related worksheets, a particular file containing such a collection, or a spreadsheet application. We use “spreadsheet” to mean a spreadsheet package when it is being used by a user. A “spreadsheet” is therefore a dynamic object, perhaps only partially completed with data, formulae, etc. A “spreadsheet” is distinguished from a “spreadsheet package” in that it has undergone interaction with the user; and a “spreadsheet application” is distinguished from a “spreadsheet” in that an application is seen as a finished product.

- user

  A user of a spreadsheet package will usually enter both code and data, and is therefore developing a spreadsheet application. This can be confusing, as traditional software engineering sees “user” and “developer” as distinct. We use “spreadsheet application developer” to mean someone (i.e. a spreadsheet user) who uses a spreadsheet package to create spreadsheet applications; and use “operator” to mean the user of the spreadsheet application, someone who only enters data into the application. The term "user" will be used in the generic sense of a user of a spreadsheet package.

- macro, macro language, procedure, function, and user-defined function

  Spreadsheet packages allow the user to define their own operations, these are defined in terms of the "macro language"; operations that can be called directly (because they have no parameters and return no values) are called "macros". The term "macros" is also used for all user-defined operations.

  The "macro language" (e.g. VBA for Excel) is used to define all user-defined operations. There are two sorts of operations: "procedures" change the spreadsheet (or its environment) and do not return a value, procedures may take parameters – "macros" are procedures without parameters; and "functions" which return a value, including "user-defined functions", which can be used in formulae (and do not change the spreadsheet.) Functions are useful in the proper development of spreadsheets as software, as a more manageable replacement for cell formulae.

- data

  We use the term "data" to refer to any numerical or string values that are displayed in cells in the spreadsheets. This includes both data entered into cells by the user (as developer or operator) and data produced by the spreadsheet application by formulae in the cells. Data that are entered into cells are of two types: *program data* entered by the developer and fixed in the finished spreadsheet application, and *input data* entered by the operator when using the application. We also distinguish two types of data produced by formulae: *output data*, which are the final results of the spreadsheet application, and *intermediate data*, which are any other values produced by formulae.

- code

  We use the term "code" as defined above, when referring to spreadsheet applications. When referring to general software engineering, "code" means source code. The message of this guide is that spreadsheet application development should treat spreadsheet code in the same way the traditional software development treats source code. Hence, we use the same term throughout.

1.3 Issues in development of the guide

1.3.1 Assumptions

We started the development of this guide with a number of working assumptions on how spreadsheet applications should be developed, as follows:

1. Using a spreadsheet package *is* software development (see, for example, [11]).

   Therefore, you must use software development techniques in order to use spreadsheet packages properly.

2. Assumption 1 is not obvious: this fact is the key to our approach to the development of this guide.

We have to explain best practice for spreadsheet application development in simple terms. We must be aware that what may be obvious to software engineers may not be obvious to spreadsheet users (who may not see themselves as software developers).

3. You can't get "inside" a spreadsheet package to find out how it works, you can only see the contents of the worksheets (see section 1.3.2).

   Software development techniques may not be easy to apply if you don't have any real conceptual model for how the spreadsheet package (supporting the application being developed) operates.

4. Risk analysis is critical to determining what level of rigour is necessary in developing and testing a spreadsheet application.

1.3.2 What's going on inside

When we discussed the problems of applying software development techniques to spreadsheet application development, we realised that it is not known by users how a spreadsheet package works. The operation of a given spreadsheet package is known only by the developers of that commercial product. For this guide, we developed a limited model of how a spreadsheet package operates.

For conventional software, a simple-minded model is based on the notion of stepping through the code line-by-line: executing a line of code involves changing the values stored in memory or jumping to another line of code; the whole process is essentially sequential. For most programming languages, there is literature explaining how code is supposed to operate; indeed for some languages there are formal standards defining the meaning of code (i.e. a language definition).

For spreadsheets, there are no standards and no obvious model of how functions in the worksheets will be executed. It is difficult to analyse the code in a spreadsheet when you don't have a sequential picture of how the code is executed. It may be difficult to get a usable printout of the code in order that code can be reviewed.

We have developed a simple execution model for how a spreadsheet package operates. We are proposing a model that we expect developers to work within, i.e. any programming that assumes more about the execution model should be avoided.

1. We assume all the relationships between cells are purely functional, i.e. each function returns the same value for given inputs: the value does not depend on other variables or values from outside the spreadsheet. Then our model is that the values in the cells are a solution of the system of equations implied by the functional relationships between cells; but we have no way of dealing with the case where this system of equations has no solution or has more than one solution.

2. We assume further that there is a strict hierarchy of cells based on their interdependencies, where the value of cells at one level only depends on cells lower in the hierarchy (no "circular references"). In this case, the value of all the cells is uniquely determined by the fixed values and we can model the execution of the spreadsheet from the bottom up.

There can still be ambiguity about which functions at a given level are executed first but the assumptions mean that the order of execution will be irrelevant.

3. However, observation suggests that when a spreadsheet is modified, only formulae and functions that depend on changed cells are re-executed. This already requires extensions to the simple execution model outlined above.
4. Our model excludes relationships between cells that are not purely functional: for instance, cells may depend on external values such as NOW() – an Excel function that returns the current time. Also, Excel has a random number function that returns a different result each time it is called.

   We have no simple model to explain when such functions are executed and re-executed, and how interdependencies between such cells are determined. Thus, we recommend that such functions are avoided if at all possible.

1.3.3 Further observations

1. Data values in spreadsheet applications come from a number of sources. We need to distinguish between program constants (provided by the application developer), input data (provided by the operator), and output data (the results of evaluations in the application).
2. The background to this work is the SS*f*M programme, which is concerned with high quality software for measurement. The spreadsheet application, in that context, must ensure the integrity of the processing from the input to the output.
3. We do not consider the problems of spreadsheet applications that are to be embedded in other applications.
4. Prototyping is a common way for spreadsheet applications to be developed: we can't ignore it and, indeed, we should consider the particular issues for software development that start from a prototyping stage.
5. We are concerned with data integrity: we are not, however, concerned with *real* security issues (e.g. hacking into the operating system via the spreadsheet application). Macros are a source of insecurity, but this disadvantage of macros is outweighed by the advantages of using them. Of course, you should avoid using spreadsheets that do not come from trustworthy sources, especially if they contain macros.
6. We do not assume it is necessarily right to develop the application as a spreadsheet application. In many cases, it will be appropriate to abandon the spreadsheet package as a development environment once the required functionality is clear. Obviously, the development of applications using approaches other than as spreadsheet applications is outside the scope of this guide.

1.3.4 Testing spreadsheet applications

NOTE

Why test spreadsheet applications?

Developers and users of spreadsheet applications think spreadsheets are straightforward and easy to understand – they do not recognise them as software. Hence they do not recognise the need to verify that the application is working as expected through testing.

Why test spreadsheets: surely they are simple and do what is expected?

1. Some spreadsheet packages have significant numerical inaccuracies and instabilities, as can be seen in [1-3].
2. Spreadsheet users create formulae and procedures to process data, so they *are* developers doing software development; and the spreadsheets are software, which needs testing/validation. The only true "users", in the software engineering sense, are those who just add data to spreadsheet applications — for which we have introduced the term "operator". Spreadsheet users do make errors in developing applications; see [11, 12].
3. Spreadsheet applications developed by copy-and-modify or by other evolutionary approaches (e.g. RAD [7], see section 2.3.2) can be poor quality software. In addition, one person's experimental calculation can become a company standard. This "evolved" software can only be deemed usable if it is tested.
4. Good practice schemes (sometimes mandated by regulation) often require standard operating procedures (SOPs) for development, testing and use of software (including spreadsheets).
5. Similarly, Quality Management Systems (e.g. the ISO 9000 series or ISO/IEC 17025) require procedures for proper development, testing and use of software.

1.3.5 Problems in testing spreadsheet applications

We have stated that spreadsheet application development is software development but there are problems in spreadsheet testing that do not occur in conventional/traditional software engineering.

- How do you input the test data and input the expected (reference) output?
- How do you compare the output with the reference output?
- How do you re-initialise the spreadsheet application before entering new test data?
- How do you deliver the test results to the user?
- Spreadsheet applications can have "wide open" user interfaces, which are hard to test.
- There is a need to supply test scripts (or other testing mechanisms) as part of the application.
- Few tools exist to support testing of spreadsheets.

You can only test spreadsheet applications effectively if the needs of testing are considered at the design and development stages. Therefore, if applications are routinely developed in an evolutionary manner, those who regularly develop spreadsheets that become the basis for applications must understand and allow for the needs of testing.

2 Key elements of spreadsheet development and testing

IN THIS CHAPTER

2. Elements of spreadsheet development and testing

2.1 Why spreadsheets are a problem

Many spreadsheet applications are developed by one person and used by that same person. If such applications are not used for a purpose that impacts the integrity of the science or the measurement or the business processes of an organisation, there may be no need to test or validate the spreadsheet application. However, many spreadsheet applications that are developed by one person for their own use, progress to become into an application that is used by others, and may be used in ways not envisaged by the original developer. The later users may use the spreadsheet application for "mission-critical" purposes, and may assume that the spreadsheet application is suitable for such use. Before the use of the spreadsheet application gets to this stage the application must be subjected to the software development process and be tested and validated for use.

A major problem with spreadsheets that are used by many users is that code will be re-used (either directly or by copy-and-modify) without considering the original purpose of the code. The original intention of the code will be lost and errors will occur, because the original developer's design (however informal) is not being followed. Spreadsheets that are allowed to develop in this way, without management control, will be impossible to validate or test properly. This evolutionary process can be controlled by documentation of design and function code, and by testing of individual functions that are going to be re-used.

It may be difficult to catch a spreadsheet in its transition from single user (acting as both developer and operator) to multi-user critical usage: business practice/procedures should be in-place to assess any software before it is used in critical applications. To be safe, some testing/validation should occur once another user, different from the original developer, starts to develop the spreadsheet further. Spreadsheets are versatile and usable tools that can be used to perform varied programming tasks by trial and error, with quick results. A balance must be struck between allowing this "experimental" use of spreadsheets and using spreadsheets as the basis for applications that are critical to the organisation. Risk analysis of the use of spreadsheets for any given area is the key to achieving the right balance.

To enable the validation/testing of spreadsheet applications it is necessary for the development process to have testability build in. Spreadsheet applications are not inherently easy to test but if the means of testing and the objectives to be tested are considered from the outset, then testing will be possible and meaningful.

2.2 Risk analysis

NOTE

The application and the development effort – fit for purpose

Risk analysis must be used to determine what level of assurance is to be achieved in the delivered application. The greater the assurance that is required the more the application will cost to develop – and the more sophisticated the tools and techniques that will be needed in development, testing and validation.

Risk analysis applied to software development aims to determine what sort of effort should go into the lifecycle of the application by looking at the impact of failure of the software in the application. This is explained in [13] in terms of measurement software levels: the greater the criticality of the intended usage, the higher the software integrity required. Spreadsheet packages and spreadsheet applications are not capable of providing the highest levels of software integrity required for safety-critical applications (e.g. where software failures can be damaging to human health).

Applied to spreadsheet applications, risk analysis suggests that for any processes that are important to your business or the integrity of your measurement, they should have some demonstrable level of software integrity. This implies proper development and testing are needed: proper development requires an understanding of the software development lifecycle; proper testing implies understanding how to perform sufficiently thorough testing of spreadsheet applications. These topics are considered in the rest of this chapter.

The risks to spreadsheet applications can be managed by having operating procedures for how the application is to be used. These procedures can restrict the freedom of users to add to and modify the spreadsheet. For more critical applications these restrictions should be imposed by the application software, and allowed modifications should be implemented by macros defined in the application.

For many applications, spreadsheet packages, although good for rapid development, offer unnecessarily powerful facilities for user interaction. The ability of users to add to and modify the applications can be useful, but it may conflict with the integrity of the data in the spreadsheet application, which may be accessed by other applications. Risk analysis and proper specification of the intended use of the spreadsheet application will determine how much user interaction with the spreadsheet application is to be allowed. Where an application may be used to supply data to another application, it is usually best for the layout and contents of the worksheets to be strictly controlled. If users want to add extra code to process the data in such an application, they should use a new spreadsheet that imports data from the original application.

2.3 Software development

2.3.1 Aims of software development

The idealised view of software development is that the user defines his requirements; these are specified as functions the software will perform, from which the code is designed, and finally the code is written; the software is then tested by producing a test plan and performing tests. This describes a purely serial process of software development that is unrealistic, impractical and not strictly necessary or even desirable, particularly for spreadsheet applications. The important results of software development are the outputs of the process (user requirements, functional specification, code design, code, test plan, tests) and the commitment of the user and the developer that they agree to and understand these outputs.

Spreadsheet applications are not usually developed in a serial manner (as described above); part of the development process is often determining what it is feasible or practical for the application to do. In this case, the user requirements to be met by the final application are not fixed at the start of the development; this will often be the case where the developer is the intended (initial) operator of the application. For spreadsheet application development, it is just as important that the various outputs are produced, so that other operators and other developers can use and maintain the application. For some spreadsheet applications that are sufficiently complex or critical, it may be appropriate to require the discipline of serial development: there are no fundamental difficulties with producing the various outputs of the software development process in the “proper” order.

An issue for spreadsheet application development, which is peculiar to spreadsheets, is the confusion between data and program. Other software starts with the code (program) and then applies the code to input data. Spreadsheet application development starts with (sample) input data and performs operations on the data. The program consists of the operations that will be performed on similar input data. As the spreadsheet is developed, cells can change from one type of data (i.e. program/input/intermediate/output) to another, and this can be a source of confusion for the developer. It is important that the distinction between data and program is clear in the finished application and in the spreadsheet application development outputs.

2.3.2 Rapid Application Development

“Rapid Prototyping” and “Rapid Application Development” (RAD) are terms used to describe the process of producing a prototype application that demonstrates the feasibility and basic functionality of an intended application with little or no regard for traditional engineering principles [7]. It is a development method that is particularly attractive for the development of spreadsheets. For the rapid prototyping phase of the software development to be useful, there must be a way to extract something (requirements, specification, design, code, test cases) from the process or from the prototype itself. Having extracted the useful elements from the rapid prototyping process, it is usually best to throw away the prototype and produce the finished product (specification, design, coding, and testing) from scratch!

It is a feature of rapid prototyping that the (initial) user and developer are the same, trying to move their work forward without having a fixed problem/requirement to overcome. The user must realise that if the spreadsheet is useful there will be other users, and there will be the need for enhancement/maintenance by other developers. Rapid prototyping must meet the needs of other (future) users by providing suitable documentation.

The development process must ensure that future maintenance and enhancement of the application are possible. It should make future development easy. Maintenance and enhancement of the application are themselves software development processes. Migration of a spreadsheet application to a new version of the underlying spreadsheet package is also part of the development process: here the crucial issue is to determine which features of the spreadsheet package have been changed and what their impact is on the application.

2.3.3 Lifecycle

A typical software lifecycle is outlined in figure 1: boxes represent the key stages, the lighter arrows show the main development path, and the darker arrows are the possible iteration loops.

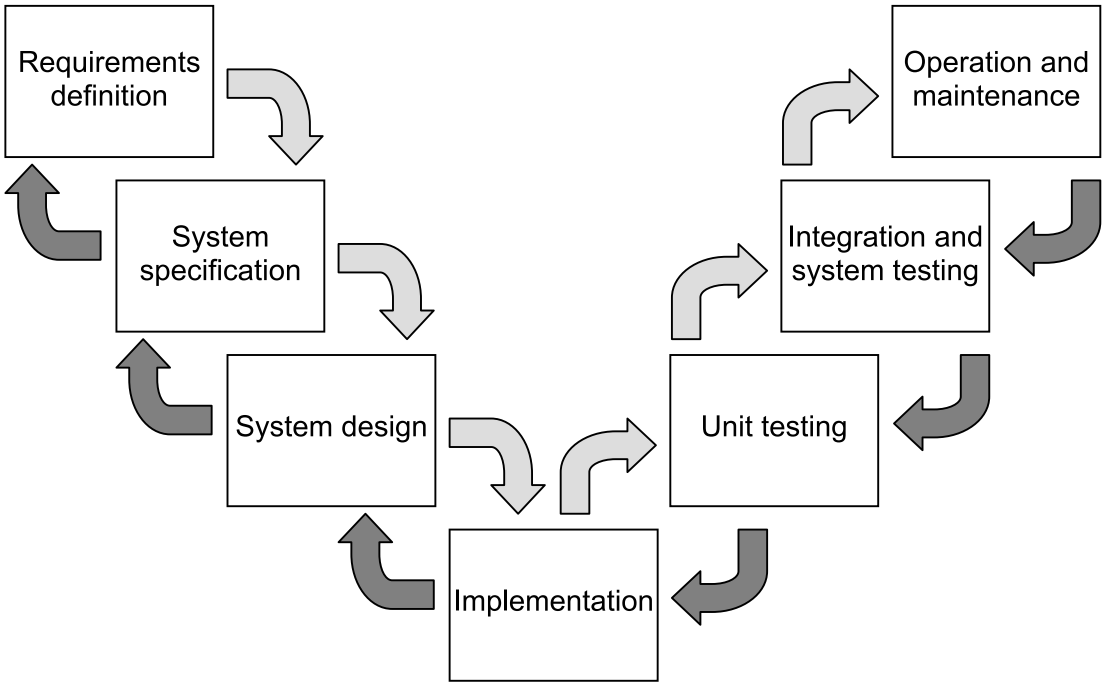


Figure 1: Typical software lifecycle

The following should be noted:

- This is an outline and for larger projects, each stage could be split into many smaller stages. For small projects, some stages could be amalgamated.
- The order of the various stages is not important but all would be expected to be performed.
- The arrows indicate a typical expected order and that the stages will be repeated if needed.
- All stages should be documented. The mapping between the various stages should be described in this documentation. For example, each requirement should be traceable to a specific part of the implementation.
- All stages of the system need to be placed under configuration management, i.e. version control.

Table 1 briefly describes the stages of the software lifecycle and includes terms used to describe these in this guide.

Stage of lifecycle	**Description**	**Other terms used in this guide**
Requirements definition	Overall description of what the system is to do.	User requirements
System specification	Detailed description of what the system is to do.	Functional specification
System design	Describes how the system will be implemented.	Code design
Implementation	Describes the implementation.	Code / Program
Unit testing	Describes the tests for parts of the implementation and the expected results.	Test plan / Testing
Integration and system testing	Describes the tests for the whole system and the expected results.	Test plan / Testing
Operation and maintenance	Describes how the system is to be used and maintained, including bug reporting.	—

Table 1: Parts of the software lifecycle

2.4 Testing of spreadsheet applications

NOTE

Why are spreadsheet applications difficult to test?

Spreadsheets are used primarily by having user data entered by hand, but software applications are tested by having the test routines enter test data automatically. So a spreadsheet application is not usually created in a way that makes automatic testing easy.

2.4.1 Problems in testing of spreadsheet applications

The issue here is *how* do you test a spreadsheet application. The usual method of testing is for (input) data to be entered into some cells and the values of other cells (the output data) checked against the expected values. The input data can be entered by hand or read from a file; the latter is necessary if testing is to be automated. Other methods of testing are possible, depending on the mode of operation of the spreadsheet application, but it is important that some form of testing is possible.

This is a central message of this guide: if a spreadsheet application is to be used in a process that is to be validated, then some form of repeatable testing must be possible; also, the design of the spreadsheet application must allow for testing a broad range of the functionality of the application.

2.4.2 Conventional approach to testing of spreadsheet applications

Conventional testing is best suited to applications that operate in distinct invocations, each invocation taking input and returning output. For a spreadsheet application, the supporting spreadsheet package is continually invoked and cells change in response to various events. It is not necessarily possible to identify a “one shot" invocation of a spreadsheet application. Where the application is one that acts on the data in input cells to produce data in output cells, one can identify an invocation as the process in which the input data is entered, the output cells are updated and the output data is read. This type of application can be tested by conventional tests that consist of input data and the known values of the corresponding output data. Furthermore, if the spreadsheet application can be invoked automatically and can read and write its data to files, then the testing can be automated. This automated testing can be performed by a test harness that controls the test data and spreadsheet application and by delivering to the user the results of running the various tests. The approach used to test the numerical functions of spreadsheet packages [1-6] has been to use macros in a spreadsheet application to read the reference data from files.

The question remains: how do you test spreadsheet applications that do not match this operational model (input cells to output cells in a distinct invocation)? The answer is that a means of testing must be developed to fit the normal operation of the application, so that the intended use of the application can be checked. Depending on the spreadsheet application and the means by which it is to be tested, it may be possible to perform the testing process automatically.

2.4.3 Built-in testing of spreadsheet applications

The cleanest approach is to have test data in worksheets in the spreadsheet application, and a macro that fills the input data cells with a test data set and verifies the contents of the output data cells against the expected results. This allows the normal operation of the spreadsheet application to be tested, and allows the test data to be kept with the application. This approach also allows for a minimal set of tests to be run each time the spreadsheet application is “started-up”, by calling a macro that runs a minimal set of tests to ensure the application is working properly, before initialising the input cells for data entry.

2.4.4 Tools for testing

Few tools exist to support the development of spreadsheet applications. Most of the existing tools for spreadsheets support the analysis of spreadsheets, otherwise referred to as auditing e.g. finding values where is should be formulas. This spreadsheet auditing would correspond to the use static analysis in software engineering.

Even fewer tools exist to support testing (sometimes confused with analysis) of spreadsheet applications. Nevertheless, this guide advocates testing as part of a software engineering approach to the development of spreadsheet applications. So, to support testing of spreadsheet applications, NPL has developed a simple test harness for Excel, called “EUnit”. EUnit has been developed to test Excel applications, based on unit testing, which has become an industry standard for object-orientated software development. Unit testing was developed initially for Java (as JUnit [10]).

EUnit for Excel consists of a workbook (spreadsheet) that is referenced from the workbook to be tested. The tests are written in VBA (the Excel macro language) and placed in modules. EUnit supplies a test harness that, when given the names of the modules, will execute all the tests in these modules, saying which tests pass or do not pass. The test harness allows a tester to concentrate on the writing of the tests rather than on how to execute the tests and record the results.

For further details see Appendix B.

3 Techniques and issues for spreadsheet applications

IN THIS CHAPTER

- Should the application be implemented as a spreadsheet?
- Which spreadsheet package should be used?
- Spreadsheet application integrity – *Check-sums*
- Ensuring testability of the application
- Layout of worksheets
- Avoiding formulae
- Coding conventions
- Macro and function design
- Use of packages and libraries
- Data checking and error handling
- Self-checking code
- Dates
- Program headers
- Comments
- Review of spreadsheet code
- Review of spreadsheet text
- Execution monitoring
- Links to other applications
- Dynamically changing the spreadsheet layout
- Understanding the effects of user interaction
- Useful generic macros
- Avoiding spreadsheets altogether

3. Techniques and issues for spreadsheet applications

3.1 Should the application be implemented as a spreadsheet?

The tabular layout of a spreadsheet gives a powerful and intuitive means of expressing user requirements and the functional specification of an application. Even if the requirements and functionality are expressed in the language of spreadsheets, this does not mean they should be implemented as a spreadsheet application – there may well be a better way. Also, if a prototype has been developed as a spreadsheet application, a spreadsheet may nevertheless not be the best way of implementing the final version of the application. (See sections 2.1, 2.2, 2.3.2.)

For the developers to answer this question, they need to draw on their experience of developing spreadsheets and their knowledge of other development environments. For any possible development environment, the developers need to consider what it can and cannot do, when they should or should not use it.

3.2 Which spreadsheet package should be used?

This may be determined by what is available to the developer and/or operators. Even if a prototype has been developed in one package, it may be right to switch to another package; e.g., because it offers different functionality or improved numerical accuracy. Again, the developers should call on their experience and knowledge of the different packages available.

3.3 Spreadsheet application integrity

Spreadsheet applications can be altered during use, either accidentally or intentionally (maliciously?), which is a problem not found with an application consisting of a compiled program. The operator should be constrained to only be able to change the input data to the application; indeed input data cells can be identified as cells the operator is supposed to change.

Techniques that can be used to prevent the operator from changing other cells include: locking formulae, protecting sheets with a password, or hiding data (e.g. closed/shrunk rows and columns, apparently blank cells, cells off-screen).

It is important that such techniques are used when appropriate to constrain what the operator can do. However, it is also important that the use of these techniques is itself validated (by testing) to avoid inadvertently over-containing the operator, and thereby preventing the operator from changing cells that they need to change.

Check-sums

Whatever is done to prevent modification of the spreadsheet application program, it is still important that there is a way to check that a spreadsheet application in use is the same as the original version. The most useful technique is to calculate a check-sum (or hash value) of the program and compare this with a check-sum of the original version. A check-sum is a way of reducing a large amount of data to a single number in such a way that the check-sum will usually change if the data changes. A check-sum is calculated using a hashing algorithm that interactively calculates a value from successive blocks of data.

The check-sum can be done internally by a macro that calculates a check-sum of all the code in the spreadsheet application (ignoring the input data cells) and comparing the check-sum with the original check-sum. Such a macro could be called each time the application is initialised. Alternatively, the check-sum could be calculated by an external tool, having first cleared the input data cells or set them all to fixed values. This operation could be part of a co-ordinated approach to configuration management (see section 2.3.3).

3.4 Ensuring testability of the application

NOTE

Think about testing as early as possible

Testing spreadsheet applications is difficult – it is less difficult if the developer thinks about testing – the earlier in the development that testing is considered the easier it will be. Remember the slogan:

Design for testability

In general, spreadsheet applications are not amenable to automated testing because they are designed for manual data input. To ensure that a spreadsheet application can be properly tested, the developer should ensure input data can be read from a file or a worksheet associated with the application (or in other ways read automatically). If data input is usually manual, it may be necessary to implement a different way of operating that is used solely for testing (see section 2.4).

3.5 Layout of worksheets

Different worksheets might be used for input, intermediate results and displaying the output. Different kinds of data within a worksheet should be in distinct areas of the worksheet, and visually distinguished (by colours or borders). The layout of each worksheet should be planned as part of the design stage; a suggested layout is in [9]. Modules should be used to define procedures or functions, where this is appropriate, to avoid complicated expressions in the formulae in cells and to minimise the use of formulae in cells.

It is best for worksheets to grow downwards rather than sideways. Cells lower down the page should depend on cells higher up, so that the logical flow is downwards. If the worksheet covers at least two full screens, the data should probably be split between two or more worksheets. More generally, new worksheets should be added if the operator will have to scroll large distances; if scrolling is necessary then this should be indicated on the worksheet.

Spreadsheets can be used to generate output in the form of printouts of worksheets and graphs. The layout of worksheets must ensure that the required data are in cells that will appear in the printed output. Graphs must also include the right information and printed graphs should not be used to obtain measurements, unless great care is taken to validate the process by which graphs are generated and printed.

It is important that when the spreadsheet application is opened by an operator, it opens at the right place: this may require the application to be saved in a particular way. The “right place” for the application to open means the most appropriate worksheet and most appropriate cells within that worksheet. This should be either the program header and instructions (see section 3.13) or the cells for input data.

3.6 Avoiding formulae

NOTE

Use a “macro language” functions instead of a cell formula

Formulae in cells are very immediate for the developer but very difficult for anyone (including the developer) to review and test.

Replacing any formula by a function call means the code can be clearly laid out in a code module – the code is easy to read and can be tested as a function.

Development, review and maintenance can be made easier if formulae in cells are replaced by function calls. This means that the “meat” of a formula appears in a function definition worksheet, so is easier to read, to comment, and to change. It is also easier to add error checking to the code, or to add diagnostic “print statements” for purposes of debugging/tracing/logging. The use of function calls, instead of formulae in cells, may also make it harder for the operator to modify the code (accidentally or deliberately) when locking the cell is not feasible, for some reason.

The argument to the function should include all the cells referred to in the original formulae, either listed as individual arguments or one argument as a range of cells that includes all the cells originally referenced. Otherwise, if the function refers to cells directly, this dependence will be hidden from the spreadsheet package and the cells that contain the function call will not be updated when the referenced cells are altered.

This example is explained in terms of an Excel spreadsheet and VBA functions

If cell A2 contains the formula “`=2*A1`”, this could be replaced by “`=DoubleIt(A1)`”, where the function is defined in table 2.

If, instead, the formula was replaced by “`=DoubleA1()`”, defined in table 3, when the value in A1 is changed then the value in A2 would not change automatically.

```
Function DoubleIt (ByVal x As Double) As Double
      DoubleIt = 2 * x
End Function
```

Table 2: An Excel/VBA function to replace the formula “= 2 * A1”

```
Function DoubleA1() As Double
      DoubleA1 = 2 * Range("A1").Value
End Function
```

Table 3: Another function to replace the formula “= 2 * A1”

3.7 Coding conventions

It is good for a development team or organisational group to develop common conventions for how applications are coded. These include rules for naming/labelling of cells and worksheets, use of particular colour or shading for given types of cells (i.e. input, output, intermediate results, documentation, and program constants).

Predetermined coding conventions should be recorded at the outset of a project; and usages that develop during development should be agreed within the development team and recorded, preferably on a worksheet of the application.

3.8 Procedure and function design

Procedure and function development should follow standard software development practice, for instance by following the development process outlined above (see section 2.3). All the inputs (i.e. cells) to a function and procedure should be identified as arguments and given when the function/procedure is called; this allows for easier re-use of functions and procedures and allows the spreadsheet package to identify dependencies between cells. As an exception, macros that are called from menus or buttons cannot take arguments.

There are two ways that function and procedures can be defined by the user:

- Most packages allow functions and procedures to be written as code in a programming language (referred to as the spreadsheet package “macro language”);
- Some packages have a *Record Macro* feature, which record the effects of keystrokes and mouse movements, and generates the corresponding source code.

The *Record Macro* technique can be invaluable as starting point but should be used as a first pass, to see how the user interaction is translated into code. Once the generated code is available, it should be examined and edited to meet the same coding conventions that apply to manually produced code. The generated code can be use for the body of functions, and procedures with parameters, not just macros. When the code from *Record Macro* has been edited, further development of that macro (or function/procedure) can only be done by further editing of the macro language code – it will not be possible to use the *Record Macro* feature for the further development of that macro.

3.9 Use of packages and libraries

The use of reliable software packages and libraries is strongly encouraged. This avoids re-coding algorithms that have already been properly coded and tested. Code in different languages can be imported by using wrapper functions and dynamically linked libraries (DLLs).

The documentation should record the source of software packages used in the application, and the version being used (including the version number of the underlying spreadsheet package).

3.10 Data checking and error handling

The extensive use of functions, rather than formulae, makes it much easier to check the consistency of the input data and to handle errors that arise. It is very important for spreadsheet applications to check that the input data has the form that the program is expecting; and that it conforms to the developers' assumptions as to what data the program should handle. Indeed, the code that checks the input data can be a definitive record of those assumptions. Errors arise when inconsistencies occur in calculations. The program should trap (where possible) errors produced by internal functions and report the errors to the operator, in as meaningful a form as possible.

3.11 Self-checking code

It is possible to trap many coding errors by building consistency checks into the spreadsheet application (see [8]). For instance, the sum of the totals of the rows of some data should be the sum of the totals of the columns. These totals should be checked and an error reported (see section 3.10) if they differ. For large programs, it may be best to collect all checks reported in one place on the spreadsheet.

3.12 Dates

NOTE

Confusions in date formats

04/05/06 is that the fourth of May or fifth of April this year, or sixth of May 2004, or something else? To avoid ambiguity we recommend using an unambiguous format for input and output: either ISO format **2006-05-04** or US military **4 May 2006**.

Spreadsheet applications usually contain date information, if only to indicate when the data was input. There are a number of problems with the use of dates in spreadsheets (and software in general), the most famous of which is (hopefully) behind us.

- Two digit and four digit years

  It is usual to write years as two digits and leave it to the context to deal with any possible ambiguities. Spreadsheet packages allow the entry of two digit years and tend to use a windowing algorithm to determine the century, i.e. they assume the year is within plus-or-minus some number of years of the current date. However, different packages and different versions of the "same" package use different algorithms, so there is no reliable way of getting a predictable four digit year when entering a two digit year. The conclusion is obvious: all dates should be entered as four digit years and all date calculations should work with four digit years.

- "Year 2000" problems

  The "Year 2000" problems are problems related to inputting, storing, manipulating and displaying dates from the year 2000 onwards. If the spreadsheet package or the underlying hardware platform has "Year 2000" problems then they should not be used for spreadsheet applications. The use of four digit years (previous bullet) should eliminate any potential for continuing "Year 2000" problems in spreadsheet applications.

- Day/Month confusion

  In some parts of the world (in particular the US) dates are commonly displayed as MM/DD/YYYY (i.e. 6/20/2000) rather than the common UK format of DD/MM/YYYY (e.g. 20/6/2000). This can cause confusion for dates in the first twelve days of each month, especially as spreadsheet packages often display US format by default. Organisations should ensure that a fixed date format is used for all its applications and there is some benefit in using less ambiguous formats, such as ISO format YYYY/MM/DD (e.g. 2000-06-20) or US military date format, e.g. 20-JUN-2000.

3.13 Program headers

The spreadsheet application as a whole (and each worksheet) should identify itself, including (as appropriate) its name, purpose, authors, dates of creation and modification, version number, instructions for use, etc. If the program is small, the program header should appear at the top of the first worksheet; otherwise, there should be a header at the top of each worksheet and the first worksheet should be dedicated to documenting the application, including the program header.

The program header should make clear the assumptions the application makes about how it will be used and what the input data is. These assumptions should be expressed in the spreadsheet or there should be a reference to the program documentation. The values of program constants should also be included with the program header, so that they can be located easily.

3.14 Comments

The code should be documented properly so that it is clear what the relevant formulae and procedures do. What is not so clear for spreadsheet applications is where the comments can appear. Some possibilities are listed below (not all of these may be possible in a given spreadsheet package):

- text in cells on a worksheet,
- text attached to cells as notes or comments,
- text embedded in procedure/function definition code.

It is not usually possible to put comments in formulae – e.g. you can't do:

```
= A1 + A2 /* sum */
```

– and such comments would probably not be readable. If it is necessary to make comments on cell formulae, it is probably best to put the formula code in a function and replace the formula in the cell by a function call (see section 3.19). It is also possible, in some spreadsheet packages, to attach notes to cells but the user will not always know to read these notes. If it is possible to have the notes appear by default, this should be done.

3.15 Review of spreadsheet code

The spreadsheet code should be reviewed against the code design but this is not so straightforward. For traditional programming, the code is available as a printout and the individual modules and functions can be read to ensure they meet the design. To review spreadsheet code, it is necessary to review the formulae in place (by selecting cells) or to obtain a printout of all the formulae in their cells. Neither of these is really practical, especially for complex cell formulae, and so complex formulae in cells should be avoided in favour of function calls (see section 3.6). Review of spreadsheet code is easier if references to other cells are written using “labels” rather than cell co-ordinates. (See section 2.3.3).

3.16 Review of spreadsheet text

The text in the spreadsheet should be reviewed to ensure it is consistent with the specification. This includes static text in cells that may be instructions to the operators, and labels for rows and columns. It is import to review all text that appears in (printed) output from the spreadsheet application: no only text in cells on the worksheets containing the output data, but also headers and footers in print layout for worksheets and graphs. The latter headers and footers are not usually visible when the spreadsheet is being developed and maintained, and it is easy to forget to update then when modifying a version of a spreadsheet application.

3.17 Execution monitoring

It is useful to be able to monitor the execution of a spreadsheet application; this can be used to supplement spreadsheet code review, to aid debugging, and to provide traceability from the execution of the application by logging significant “events”. Various mechanisms may be available in the spreadsheet package to enable logging, for instance the use of triggering events to cause messages to be printed; it will normally be possible to print logging information from macro calls.

3.18 Links to other applications

A powerful facility provided by spreadsheet packages is the ability for one spreadsheet to refer to data in another spreadsheet. This is a useful technique to allow users to perform additional calculations on data in a given spreadsheet application, without having to modify the existing application. The user sets up a new spreadsheet application, which makes a copy of the relevant data in the original application, by references, and can perform new calculations on the copied data. The user has the freedom to use the data from the original application without threatening the integrity of that application.

However, there can be problems with references to data in other spreadsheets. The main problem is maintenance: if the referenced application is changed then all references to that application may become invalid, requiring all such references to be checked and corrected. Another issue is how references are updated when data is changed in the referenced application. The user of an application needs to be sure that the values of references have been updated from the referenced application before relying on the data in the referencing application. It is probably unwise to assume that data from references will be necessarily correct if two spreadsheet applications reference one another.

It is best to use references to another spreadsheet application only if the referenced application is stable and the data in the referenced application changes infrequently. Work on the referencing application should not be done while the referenced application is changing and references to the referenced application should be updated explicitly at the start of the work. This avoids both maintenance issues and questions of when updates happen from one application to another.

3.19 Dynamically changing the spreadsheet layout

It can be necessary in the use of a spreadsheet application for the operator to change the layout of one or more worksheets, for instance to accommodate more rows of input data than the original worksheet allowed. It is not recommended to allow the operator to add new rows by hand: it will not always be easy for the operator to see what is needed and the effects of changes (and operator mistakes) may not be visible or predictable. It is better to write a macro that adds the rows/columns automatically and updates the formulae accordingly, but even this is not very satisfactory as the effects of such macros are difficult to test.

If the operator is allowed to change the spreadsheet while entering input data, the allowed user interaction should be carefully documented and included in the operator instructions. If these allowed interactions include adding rows/columns, the steps the operator is to follow should also be detailed precisely.

3.20 Understanding the effects of user interaction

When the operator is provided with macros to interact with a spreadsheet application, it can be difficult for the developer to understand the combined effects of sequences of macro calls. A powerful technique to understand these effects is to model the user interaction as a state machine.

A state machine has states, corresponding to the different configurations that the application can get into, and transitions between states, corresponding to macros that can be used to change the configuration. A diagram of the state machine should be simple enough that it can be drawn on paper as part of the documentation. The developer should use such a diagram to check the code design against the functional specification to ensure that the allowed user interaction provides all the desired functionality, and no more.

A user interaction state machine model is also invaluable in testing the user interface. The tests on the user interface should be designed so that every state in the state machine is visited. More rigorous testing would test every transition from every state: testing that allowed transitions have the right effect, and that disallowed transitions are disabled.

3.21 Useful generic macros

There are three generic macros that are useful to any spreadsheet application developer.

1. A "clear all" macro that removes all input data leaving just the program. Input data cells are left clear or replaced with default values, as appropriate.
2. A "testing" macro to read in reference data or regression test data and check known outputs.
3. An "initialisation" macro to check the integrity of the spreadsheet application (see section 3.3), and perform a self-test and then clear the input data cells.

These are generic in the sense of their functionality being application independent but they are not provided by the spreadsheet packages. The developer will need to write their own version of these macros and adapt the macros for the layout and testing infrastructure of each spreadsheet application.

3.22 Avoiding spreadsheets altogether

NOTE

Perhaps the application should not be a spreadsheet

Developing and maintaining a spreadsheet application following sound software engineering is not well supported by the spreadsheet packages and is difficult for the developer.

It may be best to regard the initial spreadsheet application as a prototype that captures the required functionality and some of the right code design, and re-implement the application as software in a real programming language, with real tool support.

The technique above "Avoiding formulae" (section 3.6) can be extended and generalised, moving code and intermediate results into functions and values passed between functions. The ultimate situation is a spreadsheet application where the output data values are calculated by a function from the input data, and all the code and intermediate data are in the functions. The spreadsheet exists only for input and output: storing and displaying input and output data.

If the language for writing functions (the "macro language"), or a related language (with similar syntax and semantics), can be used to create standalone applications, then the application can, by this route, be made standalone. This would allow an alternative mechanism for input/output to be used for the application. The application would no longer be reliant on the spreadsheet package that had been used for the development. This is available from Excel, by using the VBA procedures and functions as the basis for a standalone application in Visual Basic.

Another development technique is "spreadsheet automation", where a standalone application is developed that interacts with spreadsheet data (using spreadsheet package functions "under the hood"), while not exposing the spreadsheet to the operator. Again, this is an option for Excel, where a Visual Basic application can create and manipulate a spreadsheet object using OLE. (Microsoft's system for *Object Linking and Embedding*)

This technique offers another development route, rather than developing the application as a spreadsheet or developing the application (from scratch) in a new language. These different approaches are illustrated in figure 2.

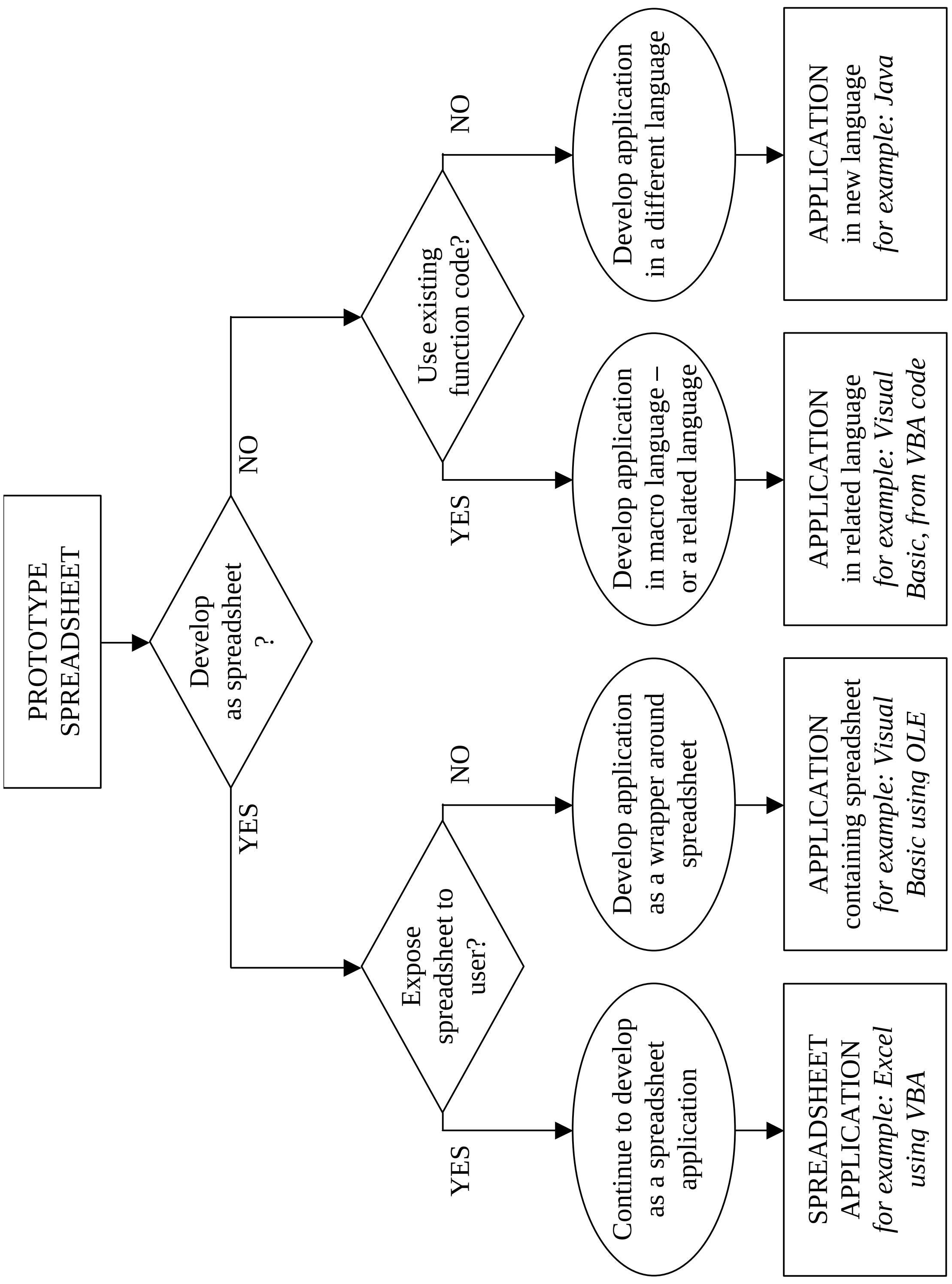

PROTOTYPE SPREADSHEET
Develop as spreadsheet ?
YES
NO
Expose spreadsheet to user?
Use existing function code?
YES
NO
YES
NO
Continue to develop as a spreadsheet application
Develop application as a wrapper around spreadsheet
Develop application in macro language – or a related language
Develop application in a different language
SPREADSHEET APPLICATION *for example: Excel using VBA*
APPLICATION containing spreadsheet *for example: Visual Basic using OLE*
APPLICATION in related language *for example: Visual Basic, from VBA code*
APPLICATION in new language *for example: Java*

Figure 2: Possible development paths from a prototype spreadsheet application

Checklists for spreadsheet development and testing

4

IN THIS CHAPTER

- What to develop
- What to validate
- What to test
- Spreadsheet organisation and layout
- Spreadsheet code review
- Organisational requirements

4. Checklists for spreadsheet development and testing

4.1 What to develop

These are questions to decide whether or not it is worth the investment of resources to develop an application as a spreadsheet application, and to determine whether it can be developed properly and validated (see sections 3.1 and 3.2). There can be good reasons to develop an application as a spreadsheet application. For instance:

- spreadsheets give an intuitive presentation of data;
- spreadsheets allow the creation of summaries and graphs (which can be especially good for measurement results and uncertainties); and
- spreadsheets are also a valuable aid to understanding, through data exploration.

This checklist is designed to ensure you do not develop an application using a spreadsheet package when it is not appropriate to do so. Spreadsheets are not the right solution for all applications even if the developer is used to using spreadsheets. Spreadsheets offer some obstacles for the sound development of applications and the decision to develop an application as a spreadsheet should be a conscious one. Some of the questions may seem obvious – but they are worth thinking about.

1. Will the spreadsheet application be an effective solution to the problem?

   If not, then it may not be best to develop the solution as a spreadsheet application.

2. Does the spreadsheet application provide functionality that is widely applicable?

   If the functionality is widely applicable, that functionality should not be implemented in a spreadsheet application – but instead should be implemented as an application that makes the functionality widely available to other applications.

3. Does the spreadsheet application perform calculations that are repeated frequently or are numerically highly complex?

   Frequent or complex numerical calculations are not very efficiently implemented in a spreadsheet package and so if they are present in the application, development as a spreadsheet application is best avoided.

4. Does the spreadsheet application provide new functionality not duplicated by an existing (properly developed and validated) application/system?

   If the functionality is available in an existing application or system, the new application should be developed to re-use the existing solution. If the existing solution can be re-used by in a spreadsheet application then the new application should not be developed as a spreadsheet.

4.2 What to validate

The following items should be validated, if applicable:

1. system requirements and functionality;
2. functions and procedures (see section 4.3: items 4 and 5);

3. algorithms and parameters (see section 4.3: items 2 and 3);
4. numerical integrity (robustness), accuracy and reliability (see section 4.3: item 1);
5. application integrity via calculating a check-sum (see section 3.3) — internal or external;
6. operational procedures;
7. training of operators;
8. change control and configuration management.

4.3 What to test

This section gives a list of the items that should be tested, if applicable.

For testing the numerical calculations:

1. test the accuracy of calculations using standard, or reference, test data (see section 3.4);
2. test the correct treatment of parameters by using test data with values of the parameters in range, on the edge of the range and outside the range of allowed/expected values (see section 3.10);
3. test the logic of calculations by using a test data set that causes conditional expressions in the code to become true and false, respectively; more rigorous testing would test all paths through the different branches in the code (see section 3.11).

For testing the user interface:

4. test the functionality of each function;
5. test the functionality of each command/button;
6. test the effects of combinations of interdependent macros (see section 3.20);
7. test the accuracy of plotted graphs (see section 3.5);
8. check that the application identifies itself in its output;
9. test the printing of each printable worksheet.

4.4 Spreadsheet organisation and layout

The following recommendations are made regarding spreadsheet layout (see section 3.5), where applicable:

1. use colouring or shading of cells to distinguish status, be wary of using colours that will give problems to colour-blind operators (particularly red and green);
2. lock cells that are not for input data;
3. protect worksheets and workbooks by password;
4. provide operator instructions on the worksheet which takes input data OR on their own worksheet (see section 3.13);

5. arrange either operator instructions or data input to appear on the visible portion of the first worksheet;
6. set the format of cells appropriately for the data that is to appear in them.

In addition, the following recommendations are made regarding the spreadsheet application interface, to be applied where applicable:

7. remove unused worksheets;
8. provide each worksheet with a meaningful name (not “Sheet1”, “Sheet2”, etc.);
9. the spreadsheet application should open at the right place (see the final paragraph of section 3.5 and items 4 & 5 above);
10. hide rows/columns that the operator need not see;
11. use separate worksheets for data that the operator need not see;
12. simplify, as far as possible, the input screen and menus;
13. disable “drag and drop”, to avoid cells being filled accidentally.

4.5 Spreadsheet code review

The following list indicates what to check in a review of spreadsheet code. All aspects should to be reviewed against the user requirements and the functional specification (see section 3.15):

1. the implementation of the calculations is correct;
2. repeated calculations have been correctly copied;
3. rounding has been done at the appropriate place in the calculation — usually in displaying the output data;
4. review function and procedure code (standard source code review);
5. review input for usability and fitness for purpose;
6. review output for readability and correctness of layout, headings, labels, etc.;
7. review instructions and documentation.

4.6 Organisational requirements

For the given criticality of the application and complexity of calculation, the organisational requirements should be specified under each of the following headings:

1. roles and responsibilities of staff;
2. documentation;
3. testing requirements;
4. version control and configuration management mechanisms;
5. audit mechanisms.

4.7 Other issues

The following is a list of miscellaneous other issues that should be considered (not all will necessarily be required):

1. document the source of equations, calculations, and algorithms;
2. can the spreadsheet application interface to other systems — e.g. Laboratory Information Management Systems (LIMS)?
3. for invalid or non-existent data, can the spreadsheet package or application distinguish between the following cell contents: clear (no data entered), “”, 0;
   e.g. when a data value is zero, as opposed to no data value having been entered.

Appendices

A

IN THESE APPENDICES

Appendix A
Procedures for spreadsheet application development

This appendix describes the procedures for spreadsheet application development in a form that can be used as a template for drawing up real procedures in a real organisation. This material was produced by reviewing quality procedures for software development against the particular requirements of this Best Practice Guide. Normal text at the beginning of each section defines what that section should contain; italicised text is suggested text that might appear in the actual procedures. We also give references to the relevant sections in the previous chapters.

The procedures should start with a short introduction:

In order to allow proper testing of spreadsheet applications, the applications shall be developed according to sound software engineering procedures. Some procedures translate easily from conventional software engineering to spreadsheet application development; but other procedures require reformulation in terms of spreadsheet concepts to make the requirements clear.

A.1 Purpose/Scope of the procedure

The purpose of this procedure is to describe best practice for the development of spreadsheet applications that will process measurement data.

- *The application needs to be able to process correct data and trap invalid data.* (See section 3.10.)
- *The application should be easy to maintain and enhance.* (See section 2.3.)
- *Further guidance should be sought if the spreadsheet application is to be used as embedded software within another system.*

A.2 Management of the spreadsheet application development process

The development process is a project, either a separate project or as part of a larger project: as such, the process should be planned and managed. There should be someone responsible for the spreadsheet application being developed; this person should approve the installation/release of the implementation. The development process should be documented and records kept. Staff employed on the development project should have the necessary training, experience, skills, and/or qualifications for the task. (See sections 2.3 and 4.6.)

A.3 Development of spreadsheet applications

A.3.1 Introduction

The introduction should explain why spreadsheet applications should be developed in accordance with this procedure (see section 2.3).

A.3.2 Specification of spreadsheet applications

The specification of a spreadsheet application consists of documents used to develop and maintain the application. These documents are the user requirements, the functional specification, the code design and the test plan.

The first three documents are in order from high level down to low level. Each document should be reviewed to see that it captures the requirements of the preceding document, as well as detailing requirements/functionality/code in sufficient detail.

The test plan validates the application against the user requirements but will go down to a lower level of the code in defining the tests. The test plan should be reviewed against the other specification documents to see that it includes tests for the important requirements/functionality/code given in the other documents.

User requirements

The user requirements document specifies why the application is needed, the problem that is solved, the functions provided to the operator, and how it should be used. One particular issue for spreadsheet applications is how input data is to be entered into the application: is it read from a file or another application, or is it entered by hand into a spreadsheet worksheet, or by some other means? (See section 2.3.3.)

Functional specification

The functional specification should spell out the algorithmic or mathematical relationship between the input and output data. It should also define how the application interacts with the environment: both the computer environment (operating system, file system, other spreadsheets, etc.) and the real world (e.g. human user interface, measurement instrumentation). *(See section 2.3.3.)*

Code design

The code design covers how the relationship between inputs and outputs will be implemented. This will be given in terms of intermediate data to be calculated, modules to perform specific calculations, and organisation of data and calculations within the spreadsheet worksheets. However, the first question for code design is should the application be implemented as a spreadsheet. *(See sections 3.5 and 3.6.)*

Test plan

The test plan should show how each testable element of the requirement, functionality and code is to be tested. The tests should show the correct operation of the application and should include data that might reveal errors in the application: e.g. invalid data that should be trapped, or values that are close to boundaries in the functionality. *(See sections 2.4 and 3.4.)*

Note on vocabulary

There may be some difficulty in finding the appropriate language to talk about elements of spreadsheet design and implementation, as this has not been the subject of technical literature. There is a need for the various application specification documents to use a common language. In the absence of any other source, they should use the terms used in the documentation of the spreadsheet package to be used for the implementation of the application (once this is known). Alternatively, the application development should maintain its own glossary/vocabulary; the glossary/vocabulary of this guide could provide a starting point.

A.3.3 Code design and implementation

This section should describe the techniques to be adopted for producing the code and structure of the application. The following should be considered:

1. Choice of implementation platform: should the application be implemented as a spreadsheet? If so, which spreadsheet package should be used?
2. Coding conventions: procedure/function design, comments, use of cell formulae.
3. Dependencies: identify libraries, error handling, and versions of packages to be used.
4. How to ensure the integrity of code: prevention and detection of changes to the application.
5. Ensuring testability of the application.
6. Structure of the application: the different spreadsheet worksheets, the layout of cells within each worksheet, program and worksheet headers.

A.3.4 Reviews

The various outputs from the spreadsheet application development process should be reviewed for consistency to ensure traceability from the user requirements to the code and the tests. The following reviews are necessary:

1. Design Review

   There should be a chain of traceability from the user requirements document, through the functional specification, to the code design. Reviewing these documents and their inter-relationships is a paper exercise. (See section 4.4.)
2. Code Review

   The code should be reviewed against the code design (see section 4.5).
3. Review of application execution: debugging/tracing/logging (see section 3.17).
4. Test review

   The test plan and test objectives should be reviewed against the specification and requirements and the individual test cases should be reviewed against the objectives for the individual tests. (See sections 4.2 and 4.3.)

A.4 Testing

This section should describe how spreadsheet applications are to be tested either externally or by self-testing (see section 3.4).

A.5 Procedures for use

This section should describe how applications are to be used (see sections 3.5 and 3.13).

The operating procedures provided in the spreadsheet application should be followed. Data should be entered in the input data cells only; no attempt should be made to alter formulae, or change procedure or function code, or call macros except as explicitly allowed in the operating procedures.

A.6 Configuration Management and Version Control

Configuration management is used to control the components of an application. This can be complicated where the components of an application are in different files, but the components of a spreadsheet application are normally the worksheets within one spreadsheet file. If this is the case, configuration management is simply a matter of version control. (See section 2.3.3.)

Version numbers are used to uniquely identify the different versions of an application. It is useful for users to be able to refer to different versions by a simple number, which should be noted in the program header and updated when new versions of the software are made available.

A.7 Documentation

User documentation is needed for operators to be able to use the application: even the original developer will need documentation to use an application that they have not used for some time. When writing user documentation, the developer should imagine someone trying to use the application in a year's time and having no one to ask for help.

Design documentation is needed for the developer to be able to maintain and enhance the application: even the original developer will need documentation to redevelop the application that they have not modified for some time. When writing design documentation, the developer should imagine someone trying to modify the application in a year's time and having no one to ask for help.

(See section 2.3.)

A.8 Use of other packages and add-ins

Spreadsheet applications may use other packages for some of their interaction with the computing environment or with the operator. Equally, they may use add-ins (such as dynamically linked libraries) to implement some of the required functionality. Where this involves the use of reliable packages and libraries, this is to be encouraged: it does not make sense to implement, in a spreadsheet package, facilities that are better implemented by an existing package or library. However, the developer should test such packages and add-ins to ensure they are fit for purpose. The documentation should record the version numbers of packages and libraries that are used and further testing should be done before modifying the spreadsheet application to use later versions.

(See section 3.9.)

Appendix B EUnit – test harness for Excel

B.1 Introduction

EUnit is a test harness to implement unit testing for Excel. EUnit is specific to Excel and is implemented as part of Excel, so it cannot be used with other spreadsheet packages. Therefore, unlike the rest of this guide, this appendix is only applicable to Excel and not to other spreadsheet packages.

A test harness is a piece of software that allows a selection of tests to be run, and the results recorded, without further interaction with the tester. Unit testing is a method of testing where each code unit (module, function, etc.) is tested, rather than testing the software as a whole. EUnit is design for unit testing of Excel applications, but can also used for system testing.

The implementation of unit testing in EUnit follows that of JUnit, for testing Java [10], including the choice of names for the testing procedures. JUnit uses Java exception handling to handle errors in tests, but it was not possible to mimic this in EUnit: instead global variables are used to record execution failures in tests.

Tests are written as procedures that ultimately call "assert" procedures that are used to indicate whether the tests pass or fail. The test procedures reside in code modules, which can also contain set-up and tear-down procedures that are run before and after each test in the module. Having written the tests, they are run using the EUnit GUI. The GUI allows you to select the tests, run the tests and display the results. The following sections describe how to write the tests and how to run them.

A beta release of EUnit is available from:
http://www.npl.co.uk/ssfm/download/documents/software/eunit/eunit-beta-1.3.zip

B.2 Writing the tests

The test procedures reside in code modules, these code module can contain other code and the test procedures are distinguished by having a name beginning with "`test`" (case does not matter). The code modules containing test can also include special procedures, "`setUp`" and "`tearDown`", to be run before and after each test in the module. The modules can contain other functions and procedures, perhaps to be used to support the test procedures, as long as their names do not match the test procedures.

B.2.1 Test procedures

All test procedures should have the following structure:

```
Public Sub testNameOfTest()
    On Error GoTo traperrors
        ' The test code goes here
    Exit Sub
traperrors:
    Call fail("Failed executing the test",
              Err.Description,
              Err.Source,
              Err.Number)
End Sub
```

The trapping of errors within a test is vital; otherwise the test harness (as a whole) could fail.

This is a simple example test procedure

```
Public Sub testMultiply()
    On Error GoTo traperrors
        Range("A1").Value = 2
        Range("B1").Value = 3
        Range("C1").Formula = "=A1*B1"
        Call assertEquals(Range("C1").Value, _
                6, , "Multiply two numbers")
    Exit Sub
traperrors:
    Call fail("Failed executing the test", _
              Err.Description, _
              Err.Source, _
              Err.Number)
End Sub
```

Only test procedures, starting with “`test`”, will be executed as tests by the harness, all other procedures will not be executed (unless called by a test).

B.2.2 setUP and tearDown

Two other procedures can be part of a test module `setUP` and `tearDown`; these are executed respectively before and after each test in a module is executed. Their structure is similar to the test procedures and is shown below:

```
Public Sub setUP()
    On Error GoTo traperrors
        ' initialisation code goes here
    Exit Sub
traperrors:
    Call fail("Failed set up", _
              Err.Description, _
              Err.Source, _
              Err.Number)
End Sub
```

```
Public Sub tearDown()
    On Error GoTo traperrors
        ' termination code goes here
    Exit Sub
traperrors:
    Call fail("Failed tear down", _
              Err.Description, _
              Err.Source, _
              Err.Number)
End Sub
```

Here is an example of the use of `setUP` and `tearDown` to save and restore cells from the worksheet that are overwritten by the test (for example, `testMuliply` defined above).

```
Option Explicit
Dim a, b, c

Public Sub setUP()
    On Error GoTo traperrors
        a = Range("A1")
        b = Range("B1")
        c = Range("C1")
        Range("A1:C1").ClearContents
    Exit Sub
traperrors:
    Call fail("Failed set up", _
              Err.Description, _
              Err.Source, _
              Err.Number)
End Sub

Public Sub tearDown()
    On Error GoTo traperrors
        Range("A1:C1").ClearContents
        Range("A1") = a
        Range("B1") = b
        Range("C1") = c
    Exit Sub
traperrors:
    Call fail("Failed tear down", _
              Err.Description, _
              Err.Source, _
              Err.Number)
End Sub
```

B.2.3 Test results

Each test must have one (or more) of the several "assert methods" that assign the test result; these are described in the next sections. If a test has failed, this is recorded as the final result of that test, although the test may continue to execute (and if in an infinite loop will remain in it!).

A test can result in one of four results, which are explained in the table below:

Possible test result	**Meaning**
Pass	The test has passed.
Fail	The test has failed at some assert method.
Abort	No assertions have been made in the test.
Error	This should not occur – error in EUnit.

Table 4: Test results for EUnit tests

B.2.4 assertTrue

Test based on a Boolean expression.

Name	**assertTrue**			
Parameters	Exp	Boolean	M	Boolean expression which if true the assertion is then true
	TestDescription	String	O	Description of the test
	Description	String	O	Usually Err.Decription
	Source	String	O	Usually Err.Source
	Number	Long	O	Usually Err.Number

Examples:

```
Call assertTrue(True, _
               "Captured a deliberate error in the test", _
                Err.Description, _
                Err.Source, _
                Err.Number)
Call assertTrue(Result, "setUP test 1")
```

B.2.5 assertEquals

Test if two values are equal.

Name	**assertEquals**			
Parameters	Expected	Variant	M	Expected result – Can be Double, Single, Integer, Long, Byte, Decimal or String
	Actual	Variant	M	Actual result – Can be Double, Single, Integer, Long, Byte, Decimal or String
	Delta	Double	O	Used when comparing Double or Single, has a default value of 4.94065645841247E-324
	TestDescription	String	O	Description of the test

Examples:

```
Call assertEquals(XY, _
                  Worksheets("testSheet1").Range("XY"), _
                  , _
                 "Test 1: integer")
Call assertEquals(5.75, _
                  Worksheets("testSheet1").Range("XY"), _
                  0.001, _
                  "Test 2: double/single")
Call assertEquals("Hello World", _
                  Worksheets("testSheet1").Range("XYs"), _
                  , _
                  "Test 4: strings")
```

B.2.6 fail

This is used when test have failed – it is equivalent to `assertTrue(False, …)`.

Name	**fail**			
Parameters	TestDescription	String	O	Description of the test
	Description	String	O	Usually Err.Decription
	Source	String	O	Usually Err.Source
	Number	Long	O	Usually Err.Number

Examples:

```
Call fail("Failed executing the test", _
          Err.Description, _
          Err.Source, _
          Err.Number)
Call fail("Fail to capture the error")
```

B.3 Running the tests

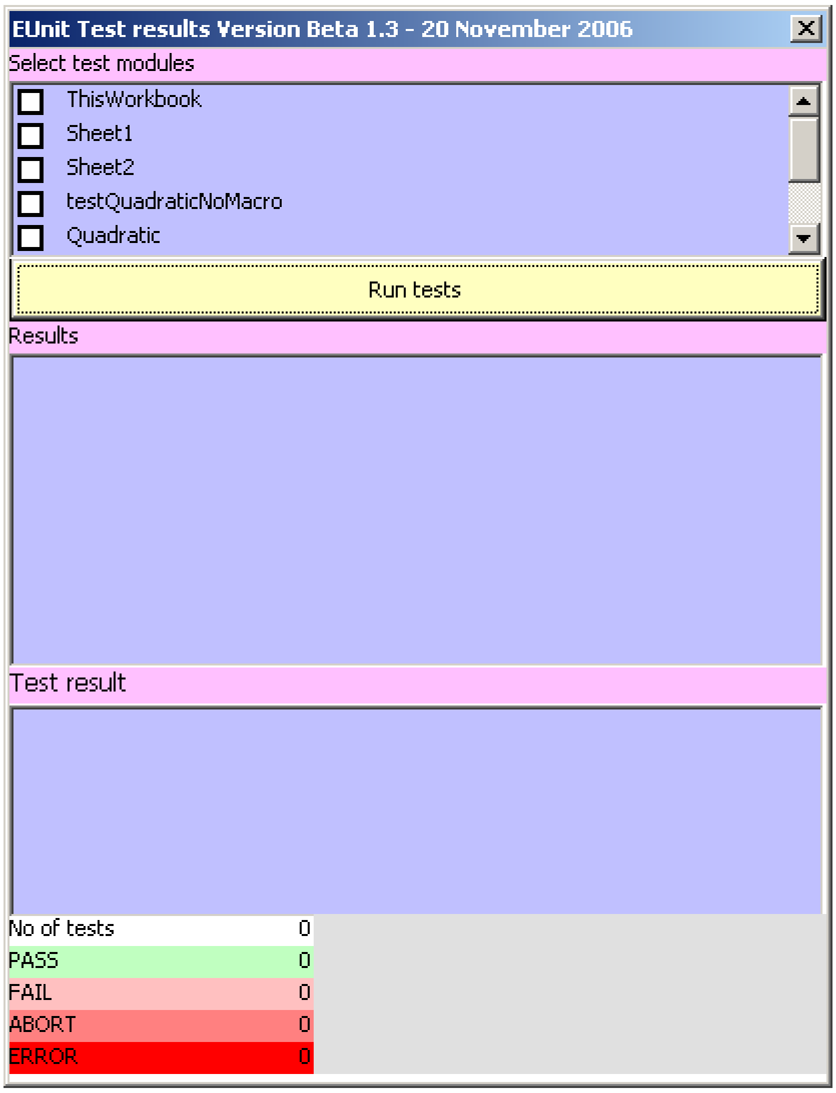
B.3.1 Installing EUnit

Before the tests can be run, EUnit must be attached to the spreadsheet application as a reference. The tests can then be executed through the supplied GUI.

From the Microsoft Visual Basic toolbar: Tools → References → Browse → EUnit.xls.

B.3.2 Using the GUI

The GUI is activated by calling the macro `EUnit.xls!showallresults`,
in the workbook with the tests in it, in which case you should see the following figure.

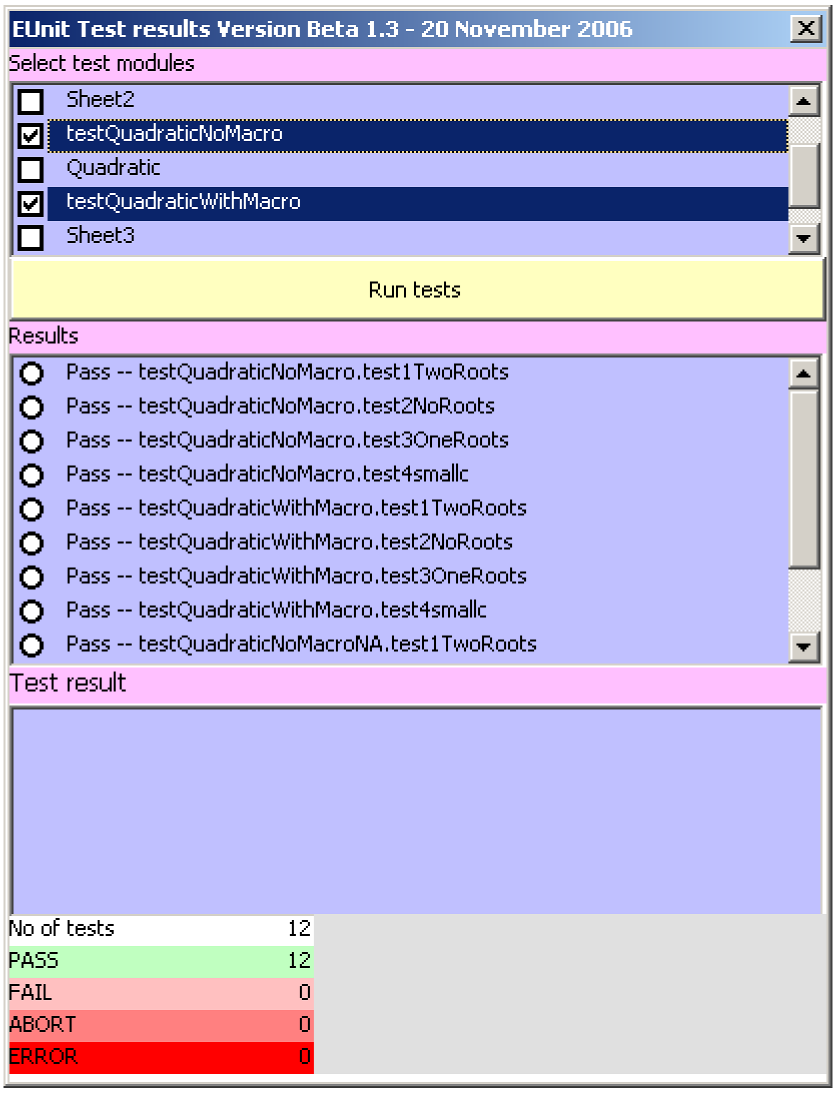


No of tests	0
PASS	0
FAIL	0
ABORT	0
ERROR	0

Figure 3: EUnit panel

B.3.3 Running tests

The top part of the GUI contains a list of all the possible code modules in the workbook. Select those with the tests and click on the yellow button. The results will appear, in the part of the GUI labelled Results, see below:


EUnit Test results Version Beta 1.3 - 20 November 2006

Select test modules

- [ ] Sheet2
- [x] testQuadraticNoMacro
- [ ] Quadratic
- [x] testQuadraticWithMacro
- [ ] Sheet3

Run tests

Results

```
Pass -- testQuadraticNoMacro.test1TwoRoots
Pass -- testQuadraticNoMacro.test2NoRoots
Pass -- testQuadraticNoMacro.test3OneRoots
Pass -- testQuadraticNoMacro.test4smallc
Pass -- testQuadraticWithMacro.test1TwoRoots
Pass -- testQuadraticWithMacro.test2NoRoots
Pass -- testQuadraticWithMacro.test3OneRoots
Pass -- testQuadraticWithMacro.test4smallc
Pass -- testQuadraticNoMacroNA.test1TwoRoots
```

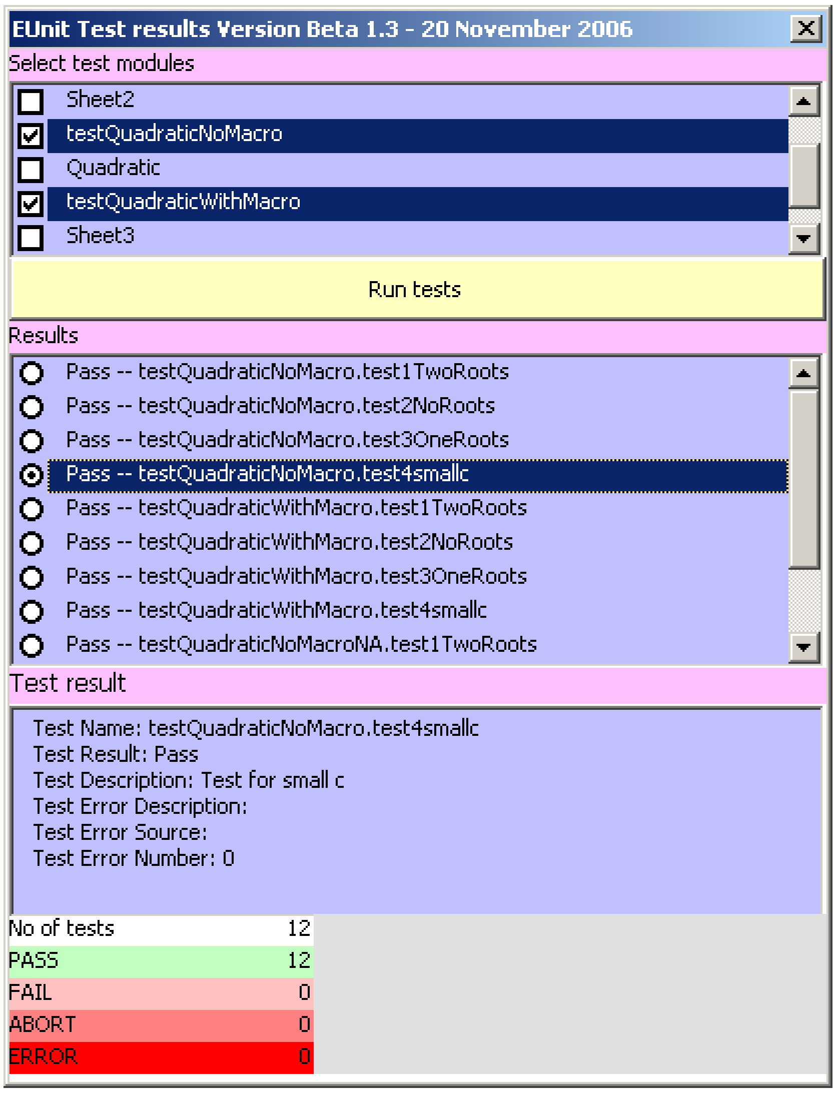
Test result

No of tests	12
PASS	12
FAIL	0
ABORT	0
ERROR	0

Figure 4: EUnit test results

B.3.4 Displaying details of tests

For any test further details can be obtained by clicking on it:



- [ ] Sheet2
- [✓] testQuadraticNoMacro
- [ ] Quadratic
- [✓] testQuadraticWithMacro
- [ ] Sheet3

```
Pass -- testQuadraticNoMacro.test1TwoRoots
Pass -- testQuadraticNoMacro.test2NoRoots
Pass -- testQuadraticNoMacro.test3OneRoots
Pass -- testQuadraticNoMacro.test4smallc
Pass -- testQuadraticWithMacro.test1TwoRoots
Pass -- testQuadraticWithMacro.test2NoRoots
Pass -- testQuadraticWithMacro.test3OneRoots
Pass -- testQuadraticWithMacro.test4smallc
Pass -- testQuadraticNoMacroNA.test1TwoRoots
```

```
Test Name: testQuadraticNoMacro.test4smallc
Test Result: Pass
Test Description: Test for small c
Test Error Description:
Test Error Source:
Test Error Number: 0
```

No of tests	12
PASS	12
FAIL	0
ABORT	0
ERROR	0

Figure 5: EUnit test detail

References

These are books and articles we found useful and relevant. They cover the application of software engineering practice to spreadsheet application development and rapid application development.

1. Cook, H.R., M.G. Cox, M.P. Dainton, and P.M. Harris. *A methodology for testing spreadsheets and other packages used in metrology. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme.* NPL Report CISE 25/99, NPL, September 1999. http://www.npl.co.uk/ssfm/download/#cise25_99
2. Cook, H.R., M.G. Cox, M.P. Dainton, and P.M. Harris. *Testing spreadsheets and other packages used in metrology: A case study. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme.* NPL Report CISE 26/99, NPL, September 1999. http://www.npl.co.uk/ssfm/download/#cise26_99
3. Cook, H.R., M.G. Cox, M.P. Dainton, and P.M. Harris. *Testing spreadsheets and other packages used in metrology: Testing the intrinsic functions of Excel. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme.* NPL Report CISE 27/99, NPL, September 1999. http://www.npl.co.uk/ssfm/download/#cise27_99
4. Cox, M.G., M.P. Dainton, and P.M. Harris. *Testing functions for the calculation of standard deviation.* NPL Report CMSC 07/00, October 2000.
5. Cox, M.G., M.P. Dainton, and P.M. Harris. *Testing functions for linear regression.* NPL Report CMSC 08/00, October 2000.
6. Cox, M.G., P.M. Harris, E.G. Johnson, P.D. Kenward, and G.I. Parkin. *Testing the numerical correctness of software.* NPL Report CMSC 34/04, January 2004.
7. DACS, *Rapid Application Development (RAD).* Software Tech News, 1998. **2**(1).
8. ICA. *Spreadsheet Design.*, Institute of Chartered Accountant in England and Wales, 1994.
9. McDowall, R.D., C. Burgess, and B. Hardcastle, *Spreadsheets: heaven or hell.* Scientific Data Management, 1999. **3**(3): p. 8-17.
10. Object Mentor. *JUnit*: 2006. http://www.junit.org
11. Panko, R.R. and R.P. Halverson Jr. *Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks*. in *29th Annual Hawaii International Conference on System Sciences*. 1996. Hawaii: IEEE.
12. Panko, R.R., *What we know about spreadsheet errors.* Journal of End User Computing, 1998. **10**(2): p. 15-21.
13. Wichmann, B.A., R.M. Barker, and G.I. Parkin. *Validation of Software in Measurement Systems.* Software Support for Metrology Best Practice Guide No. 1, NPL, March 2004. http://www.npl.co.uk/ssfm/download/bpg.html#ssfmbpg1