NPL
National Physical Laboratory

Report

NPL REPORT
DEM-ES 022

Best Practice in Software Development – A Case Study illustrated by the Softgauges Project

P M Harris, G I Parkin and
M J Stevens

Not restricted

March 2007

dti
National Measurement System

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

Switchboard 020 8977 3222 | **NPL Helpline** 020 8943 6880 | **Fax** 020 8943 6458 | **www.npl.co.uk**

Best Practice in Software Development – A Case Study illustrated by the Softgauges Project

P M Harris, G I Parkin and M J Stevens
Mathematics and Scientific Computing Group

March 2007

ABSTRACT

This report outlines a case study in the application of SS*f*M Best Practice Guide No. 1, "Validation of Software in Measurement Systems". The guide provides recommended techniques for those developing software for measurement systems to ensure the software is fit-for-purpose, and to meet the requirements of ISO 9000. The case study shows the practical application of the guide to the "softgauges" project, a project to develop reference software for the determination of surface texture parameters.

The report also shows how the same project adheres to the NPL software procedure, NPLQP/M13, which is part of the NPL quality system. The procedure assures that software development at NPL meets the requirements of ISO 9000. This report provides an illustrated use of the guide and the NPL procedures, to assist those who plan to use them as part of their software development.

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 Policy Unit)

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

Contents

List of Tables

List of Figures

1. Introduction

This report outlines a case study in the application of SS*f*M Best Practice Guide No. 1, "Validation of Software in Measurement Systems" (BPG1) [16]. The BPG1 provides recommended techniques for those developing software for measurement systems to ensure the software is fit-for-purpose, and to meet the requirements of ISO 9000. The case study shows the practical application of the BPG1 to the *softgauges* project, a project to develop reference software for the determination of surface texture parameters. The aim of the case study is to assist those developing software, explaining how the software can be made fit-for-purpose.

The report also shows how the same project adheres to the NPL software procedure, NPLQP/M13, which is part of the NPL quality system. The procedure assures that software development at NPL meets the requirements of ISO 9000. This report provides an illustrated use of the BPG1 and/or the NPL procedures, to assist those who plan to use them as part of their software development.

The report describes the *softgauges* project in section 2, and describes the Best Practice Guide No.1 methodology in section 3. The implementation issues are discussed in section 4, and section 5 describes how the validation techniques were applied. Section 6 describes the application of the NPL quality system. The appendices contain the MATLAB specification and the Java implementation.

2. Description and purpose of the *softgauges* project

The *softgauges* project involved developing and disseminating a software measurement standard, in the form of reference software (a type F2 measurement standard [6]) and reference data (a type F1 standard). The software comprises a collection of routines, each of which implements a different measuring gauge, or "soft gauge". The software is used for calibrating the software component of a surface measuring system and testing its numerical correctness. Consequently, the most important consideration in the design and implementation of the software standard was its numerical correctness. The target software components are tested by comparing the results it delivers when operating on a data set with the results produced by the (reference) *softgauges* software when operating on the same data set.

In addition to the soft gauges themselves, the project produced mathematically well-defined and unambiguous specifications of the profile parameters addressed by the soft gauges. The internet was used as the primary mechanism for disseminating the results [9].

3. Application of Best Practice Guide No. 1

This BPG1 concerns the best practices to be followed when producing software that forms part of a measurement system. The aim of BPG1 is to have software developed using techniques that will ensure the software is fit-for-purpose; without the cost of the techniques being prohibitive in comparison with costs of software failure.

The BPG1 advocates a risk-based approach to software validation described by the following steps:

1. A *risk assessment*, the purpose of which is to make an objective assessment of the likely risks associated with a software error. The assessment considers the risk factors (a) criticality of usage, (b) legal requirements, (c) the impact of complexity of control, and (d) the complexity of data processing.
2. Assigning a *measurement software level* indicated by the results of the above risk assessment.
3. Applying *software validation techniques* indicated by the assigned measurement software level.

This case study is based on Best Practice Guide No. 1 Version 2.1, dated March 2004. A new version is now available (Version 2.2, dated January 2007), in which there are a number of changes to avoid confusion with IEC 61508: in particular "Measurement Software Level" has been replaced by "Measurement Software Index" (MSI).

3.1 Softgauges Risk Analysis

3.1.1 Criticality of usage

The BPG1 gives four categories on an increasing scale of criticality. These are:

1. **Critical** – correct operation of the measurement system software is considered sufficiently important that appropriate software validation should be undertaken, without being in any of the higher categories below.
2. **Business critical** – failure could cause serious financial loss, either directly or indirectly.
3. **Potentially Life-Critical** – failure indirectly puts human life at risk or directly puts human health at risk. Such software is considered safety-critical in the context of IEC 61508 [5].
4. **Life Critical** – failure directly puts human life at risk, and hence the software is safety-critical.

The criticality of usage of the *softgauges* software is **Business critical**.

3.1.2 Legal requirements

Many measurement systems are used in contexts for which there are specific legal requirements, and where a system malfunction could have serious legal consequences.

The *softgauges* software is not used in such a context, and so does not have any legal requirements.

3.1.3 Complexity of control

This is classified in four categories of increasing complexity. The BPG1 provides examples to assist the user in selecting the appropriate category:

1. **Very simple** – e.g. when the system detects if a specimen is in place, either by means of a separate detector, or from the basic data measurement reading. The result of the detection is to produce a more helpful display read-out.

2. **Simple** – e.g. temperature control undertaken so that temperature variation cannot affect the basic measurement data.
3. **Modest** (Moderate)– e.g. if the system takes the operator through a number of stages, ensuring that each stage is satisfactorily complete before the next is started. This control can have an indirect effect upon the basic test/measurement data, or a software error could have a significant effect upon that data.
4. **Complex** – e.g. the software contributes directly to the functionality of the measurement system. For instance, if the system moves the specimen and these movements are software controlled and have a direct bearing upon the measurement/test results.

The complexity of the *softgauges* software is **Simple**.

3.1.4 Complexity of data processing

The BPG1 provides four categories of increasing complexity with helpful notes to enable the user to select the right one:

1. **Very Simple** – the processing is a linear transformation of the raw data only, with no adjustable calibration taking place.
2. **Simple** – simple non-linear correction terms can be applied here, together with the application of calibration data. A typical example is the application of a small quadratic correction term to a nearly linear instrument, which is undertaken to obtain a higher accuracy of measurement.
3. **Modest** (Moderate) – well-known published algorithms are applied to the raw data. It is assumed that the algorithms are numerically stable and there is evidence to support this.
4. **Complex** – anything else

The Softgauges software falls into the **Moderate** category.

3.2 Calculating the Measurement Software Level

The Measurement Software Level, or MSL for short, is an integer in the range 0 to 4, and is calculated from the criticality of usage, the complexity of control, and the complexity of data processing. The BPG1 provides a simple table showing the level for each possible value of the three risk factors. Table 1 is an extract of the relevant section of the table, and shows that the *softgauges* software has an MSL of 2.

Criticality of Usage	Complexity of Processing	Impact of Complexity of control
		Simple
Business Critical	Very Simple	1
	Simple	1
	Moderate	**2**
	Complex	2

Table 1: Measurement Software Level

3.3 Selection of Software Validation Techniques and Tools

The purpose of using software validation techniques is to mitigate the risks. The BPG1 says: "If the *supplier* can assure the *user* that appropriate techniques have been applied, then the use of the measurement system in the agreed role can be justified."

While ISO 9001 has general requirements, the BPG1 [16, Chapter 12] recommends specific techniques. In all, twenty techniques are discussed, with advice given on their scope, and examples of their application. A simple selection table is provided in the BPG1, in the form of a table [16, Table 11.2]. It is a simple matter to select the required techniques and read up on them. Table 2 shows all the techniques, with ticks against those relevant for MSL 2:

BPG 1 reference	Recommended Technique	MSL 2
12.2	**Review of informal specification**	✔
12.3	Software inspection of specification	✔
12.4	**Mathematical specification**	✔
12.5	Formal specification	
12.6	**Static analysis**	✔
12.6	**Boundary Value Analysis**	✔
12.7	**Defensive programming**	✔
12.8	**Code review**	✔
12.9	Numerical stability	✔
12.10	Microprocessor qualification	
12.11	Verification testing	
12.12	Statistical testing	✔
12.13	Structural testing	
12.13	**Statement testing**	✔
12.13	**Branch testing**	*
12.13	**Boundary value testing**	✔
12.13	Modified Condition/Decision testing	
12.15	Accredited testing	✔
12.16	**System-level testing**	✔
12.17	Stress testing	✔
12.18	Numerical reference results	✔
12.19	**Back-to-back testing**	✔
12.20	Comparison of the source and executable	

Table 2: Software Validation Techniques

Sixteen software validation tools are recommended for software with an MSL of 2.

Eleven techniques (set in bold in table 2) were chosen to apply to the *softgauges* software, including one technique only recommended at higher levels (*). These are shown in table 3, together (where appropriate) with the tool used to apply the technique.

Technique	**Tool(s) applied**
Review of informal specification	-
Mathematical specification	MATLAB [15]
Static analysis	Java compiler 1.4.2_4 [14] Checkstyle 3.3 [2, 4]
Boundary value analysis	-
Defensive programming	-
Code review	Checkstyle 3.3 [2, 4]
Statement testing	Clover 1.3_02 [3]
Branch testing	Clover 1.3_02 [3]
Boundary value testing	-
System level testing	-
Back to back testing	MATLAB [15]

Table 3: Techniques and Tools

It is widely thought that errors in the specification of software are the most common and most expensive to correct. In consequence, techniques that are likely to detect faults in the specification prior to investing effort in the implementation are very worthwhile. In this work, a *mathematical specification* [16, Section 12.4] of the calculation to be carried out by each soft gauge was prepared. The specification provided a mathematical statement of the algorithmic steps required to calculate the outputs of each soft gauge from its inputs. The specification also defined any constraints on the values of the inputs (*boundary value analysis* [16, Section 12.6]). An independent *review of the specification* [16, Section 12.2] was performed and included consideration of the *numerical stability* [16, Section 12.9] of the specified algorithms. The specification was considered sufficiently simple for a *software inspection of the specification* [16, Section 12.3], i.e., use of a formal process of reviewing the development of an "output document" (such as software code) from an "input document" (such as a specification), to be unnecessary.

Defensive programming [16, Section 12.7] was used throughout the software coding. Although essentially a design technique, defensive programming has an impact on the validation of software. The technique involves including code to check explicitly (and dynamically) whether a pre-condition on, e.g., a program unit, is satisfied.

The developed code was subject both to an independent *code review* [16, Section 12.8] and *static analysis* [16, Section 12.6]. Code review is applied to ensure the readability and maintainability of the source code, as well as potentially to identify software errors. Static analysis is used to determine properties of the software without execution, e.g., to identify variables that are not explicitly declared. In this work, the techniques were applied as a manual process supported by tools. For example, the tool Checkstyle [2, 4] was used to compare the code against guidelines [12] for Java programming provided by SUN.

Component testing is a basic software engineering technique that includes statement testing, branch testing and boundary value testing. *Statement testing* [16, Section 12.13] concerns the statement coverage achieved by the test cases. If a statement is not executed, then a reason for this should be documented. Note that defensive programming (see above) may introduce statements that are never executed (because pre-conditions are always met). In this work, the tools JUnit [10] and Clover [3] were used to support the application of these validation techniques.

Branch testing [16, Section 12.13] was also carried out although it is not a requirement at this MSL. It involves testing the coverage of different branches from decision points.

The conditions given in the software specification on the input values to the *softgauges* were used to define test cases for the behaviour of the software at the boundaries defined by these conditions (*boundary value testing* [16, Section 12.13]). Testing involved using each soft gauge as a "black-box" to test a known reference implementation of software to solve the problem addressed by the soft gauge (*system testing* [16, Section 12.16]). The reference implementation (MATLAB) provided an alternative means of calculating reference results (*numerical reference results* [16, Section 12.18]) against which the output of the soft gauge was compared (*back-to-back testing* [16, Section 12.19]).

Accredited testing [16, Section 12.15] was not appropriate, as no standard has been defined against which to test the software.

Statistical testing [16, Section 12.12] is used to investigate the reliability rate for the software system in the case that a statistical distribution for the inputs to the software is known. *Stress testing* [16, Section 12.17] involves producing test cases that are more complex than those that are expected to arise in practice. Application of these techniques was considered unnecessary for the complexity of the software of concern here.

4. Languages for specification and code

For convenience, MATLAB [15] was used to specify the project; and the MATLAB code could be executed to perform the required calculation, so the specification also acted as prototype code. But the MATLAB specification runs too slowly for practical use and, instead, the software was implemented in Java [14]. There were a number of issues with both MATLAB and Java and more particularly in the interplay between the two languages.

- Although the final implementation was all in Java, at an intermediate stage in the code development, MATLAB was used to call Java routines.
- Java can evaluate expressions either in *FP-strict* or *non-FP-strict* mode (the default). In FP-strict mode, all intermediate values of a computation are properly rounded and held in the appropriate IEEE 754 format, with the result that the program will produce exactly the same numerical results on all Java virtual machine implementations.

  FP-strict mode was used in the *softgauges* project. The mode is set by including the word `strictfp` in the class definition. A StrictMath library of mathematical functions was also used, as this also works in FP-strict mode [11]
- When converting the MATLAB code to Java, care had to be taken with:
  - Array bounds, which start from 1 in MATLAB, but from 0 in Java.
  - Scoping of variables as these are be created in inner blocks and then seen in outer blocks in MATLAB.

  - Passing of parameters in MATLAB is truly by value whereas in Java an object is by value only in terms of its reference (pointer) rather than its contents.
  - MATLAB can extend arrays dynamically, Java cannot.
  - MATLAB has very weak typing.

- The Java language allows for functions to be passed as parameters (encapsulated in a object), and advantage was taken of this facility as the MATLAB specification used this.

5. The Software Development Life-cycle

For the purposes of quality assurance, software can conveniently be broken down into distinct phases. Each phase can be documented, and the documents can be produced as evidence that the quality system is being followed. At each phase we show which of the relevant techniques (selected in section 3.3) were applied.

The *softgauges* project comprises a number of routines that were all handled by the quality system in the same way. One of these, `evalparama`, has been singled out as a good example for this case study.

5.1 Requirements

A requirements document is the first thing that must be written before work can begin on developing any software. It will include an outline of the application and the platform it will run on. The document must be approved by the customer – it may even be provided by the customer. Furthermore, as the project develops, the requirements document may be updated, particularly if the requirements gathered originally were incomplete, unreasonable or infeasible.

Typically, the document would cover:

- why the system is needed
- problem(s) to be solved
- outline of functions it will carry out
- intended environment
- acceptance criteria
- cut off date for bug reporting
- maintenance of software

For the Softgauges project, all information needed for the requirements document was contained in the project proposal. No additional work had to be done here: the proposal was used as a reference for the requirements.

5.2 Functional Specification

This is more detailed than the requirements document and provides enough information for a competent person to the implement the project in software. It describes what is to be implemented without dictating how it is to be done. It will cover details of how the end-user is to interact with the system, and hence may need

to be written in collaboration with the customer. The document may be incomplete at the start of the project, but it will evolve with each cycle of the development until eventually it meets all the functional requirements of the project.

Typically, the document would cover:

- inputs
- outputs
- functions
- performance
- usability
- security
- operational platform
- special hardware requirements

For measurement software, the functional specification would normally consist of a mathematical specification. In *softgauges*, it is a MATLAB implementation of the code. This provides a rigorous specification that is also a working program. As an example, the MATLAB for the Java routine `evalparama` is given here.

5.2.1 The main routine

The full MATLAB code for the main routine that calls the `evalparama` Java function (here, in the MATLAB version, called `EvalParamRa`) is given in appendix A.1.

The routine consists of five tasks, the third of which includes the call to `EvalParamRa.`

1. Evaluate the profile to be analysed

```
[ z, x, SampInd, CN, SampLen, EvalLen ] =
   EvaluateProfile(ParamStr, p, DeltaX, LambdaC);
```

2. Fit natural cubic spline interpolant to the profile points (augmented with zero-crossing points and sample length endpoints).

```
[ z2 ] = NatSplInt(z, x);
```

3. For each sample length evaluate the surface texture parameters in terms of interpolant:

```
for k = 1 : CN
   ParamRa(k) = EvalParamRa( z, x, z2, SampInd, k, SampLen);
end
```

4. Evaluate the aggregated surface texture parameters.

```
ParamVal(1) = mean(ParamRa);
```

5. One of the surface texture parameters is defined over the whole profile rather than in sample lengths:

```
ParamVal(10) = EvalParamRt0( z, x, z2, SampInd, CN, SampLen);
```

5.2.2 `evalparama`

The full MATLAB code for the function `evalparama (EvalParamRa)` is given in appendix A.2. The function evaluates parameters *Pa*, *Ra* and *Wa*.

First the function accumulates estimates of integrations over each interval making up the sample length, each estimate obtained by Romberg integration of order 4

(Simpson's rule). The parameter is evaluated by summing the integration estimates and dividing by the sample length.

`EvalParamRa` calls `RombInt`, and passes the function `FuncRa` as a parameter. The MATLAB code for `RombInt`, which implements Romberg's method of order 2K for the integration of a function, is given in appendix A.3. The code for `FuncRa` is given in appendix A.4. This evaluates, for parameters Pa, Ra, and Wa, the absolute value of the natural cubic spline interpolant. This it does by calling the function EvalSplint – the MATLAB for this appears in appendix A.5.

5.3 Design

A software design document explains exactly how the details outlined in the functional specification are implemented. If implementation details change during development, the document will need to be kept up-to-date. It should form a useful part of the system documentation.

Typically, the document would cover:

- program structure
- data structure
- mapping to functional requirements
- coding conventions
- software tools used

The *softgauges* project was implemented in Java. The language is object-oriented, which makes it convenient to impose a structure on the code. The project can easily be divided into sets of routines that belong together, and these sets can be implemented as objects, with the routines forming the methods.

A software development tool BlueJ [7] was used as a development environment for writing the Java code. BlueJ provides a pictorial display of the structure of the code, and this forms an ideal means of documenting the design. Since this is generated automatically, it necessarily mirrors the actual design of the implementation, and keeps in line with it whenever changes are made. By contrast, handwritten documentation is time-consuming to produce and it is easy to forget to keep it up-to-date when changes are made.

The BlueJ design diagram is shown in figure 1. The darker coloured boxes at the top of the picture represent packages that contain collections of related classes used by the project. The lighter coloured boxes represent classes or interfaces, each open arrowhead showing where one class calls another class, and each closed arrowhead indicating where a class implements an interface or extends a class. Thus, it is easy to see that the `SoftGauges` class calls the `EvalParama` class, while the latter calls the `SoftGaugeResults` class and implements the `SoftGaugeFunction` interface. The `SoftGaugeFunctionAdapter` class has been introduced because the classes that extend it do not need to access all the methods provided in the `SoftGaugeFunction` interface.

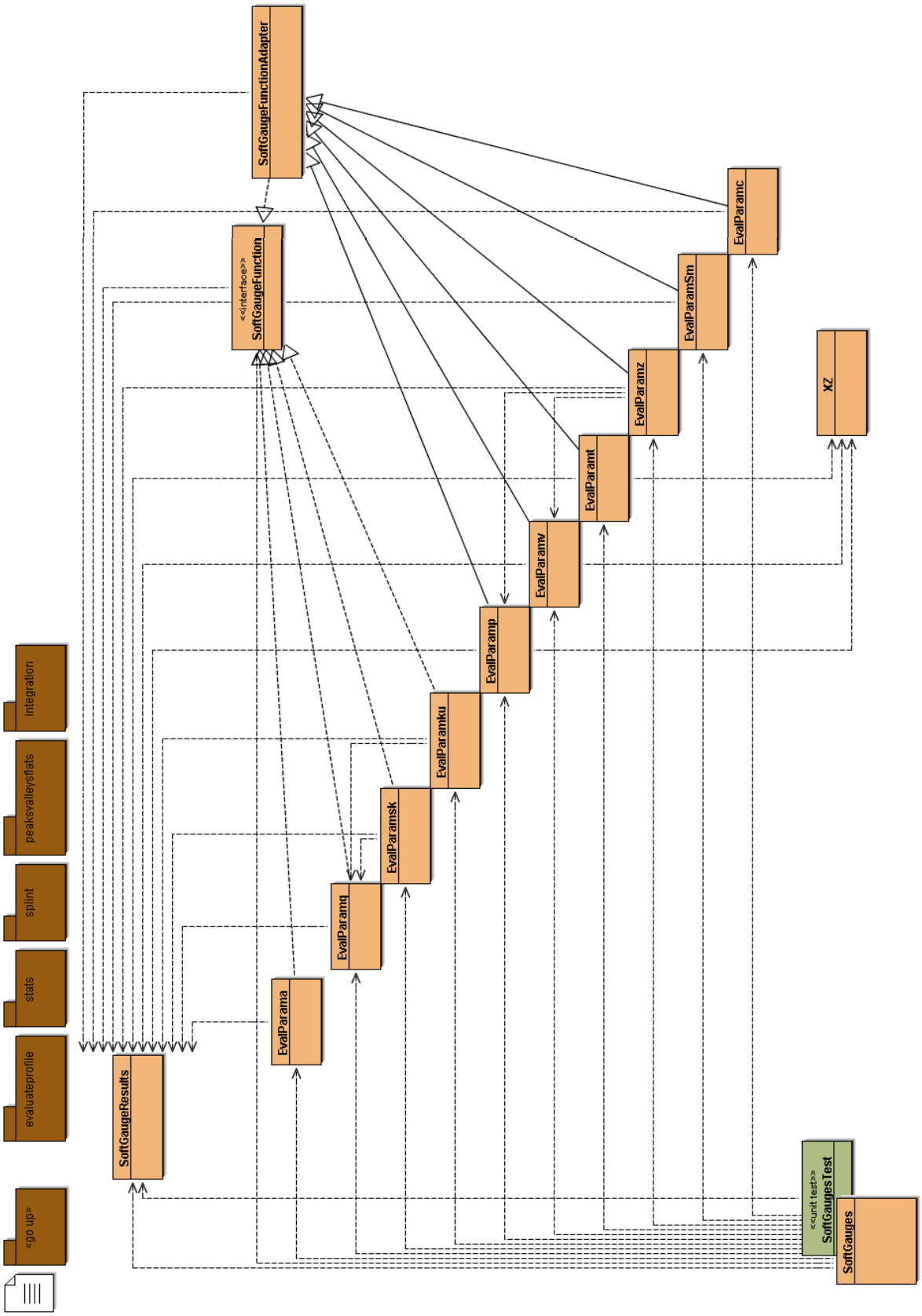

Figure 1: *BlueJ* Design Diagram

5.4 Implementation

5.4.1 Java code

The Java code corresponding to the MATLAB functional specification is listed in Appendix B, starting with the `SoftGauges` class. Table 4 shows the mapping between the MATLAB functions and Java methods. The Java has been written to follow the MATLAB specification as closely as possible. Unlike the MATLAB code, the Java includes error checking, constraint checking and a method to derive the units of the various parameters (`SoftGauges.ParameterUnit`).

MATLAB function	Equivalent Java method
`SoftGauge`	`SoftGauges.softGauge`
`EvalParamRa`	`EvalParama.evalParam`
`RombInt`	`Integration.rombInt`
`FuncRa`	`EvalParama.parameterFunction`
`EvalSplInt`	`SplInt.evalSplInt`

Table 4: Mapping MATLAB functions to Java methods

5.4.2 Defensive programming

The defensive programming in the Java code was mainly included to confirm the assumptions of those writing the code. These were assumptions about the nature of the surface textures and assumptions about values from interfaces: either values of parameters from the GUI or the behaviour of library functions.

As with the error handling, this code was not included in the MATLAB specification.

5.5 Static Techniques

5.5.1 Static analysis

This was accomplished by using checks provided in the Java compiler [14], and Checkstyle [2, 4].

5.5.2 Code review

This was conducted by independent persons, covering correctness, readability and maintainability.

5.6 Testing

5.6.1 Back-to-back testing

The testing tool JUnit [10], which is linked into the BlueJ development system was used to compare results produced by the Java implementation with the output from the MATLAB code when operating on the same data.

Using JUnit, a test class can be created in BlueJ with the click of a button, and a drop-down list is then provided for the user to select the Java method of interest. The user is prompted for the parameters and the expected results. A test method is generated

that produces a pass or failure indication when the actual results are compared with the expected results.

Exhaustive testing of all methods is possible, though it is a time-consuming task to generate all the tests. Parameters that lie within, on the boundaries, and outside the working range can be given in order to check that methods work when they should work and fail with appropriate error messages when they should fail. Once written, the tests can simply be repeated any time a change is made to the code.

Methods `evalParama` and `RombInt` are not tested as separate units, but the `SoftGauges` class is subjected to sixteen tests, the first of which exercises `evalParama` which in turn calls `RombInt. SplInt` also has its own set of tests.

The testing of the `SoftGauges` class is performed in method `runPRWforSoftGauge`, shown in section B.4.

- This reads data from a file `fileIO.readfiles.filereaders.readdatalines`,
- calls `SoftGauges.softGauge`,
- and checks the results (`assertEquals`).

5.6.2 Statement and branch testing

This was a code coverage analysis, to check that all parts of the code were exercised at run-time. It was tested using the *Clover* tool [3]. The test data achieved nearly 100% code coverage (also true for the MATLAB implementation).

Figure 2 shows the code coverage report for the Java implementation, produced by the Clover package.

5.6.3 Stress testing

The Java implementation was executed on a large test data set containing 200,000 points.

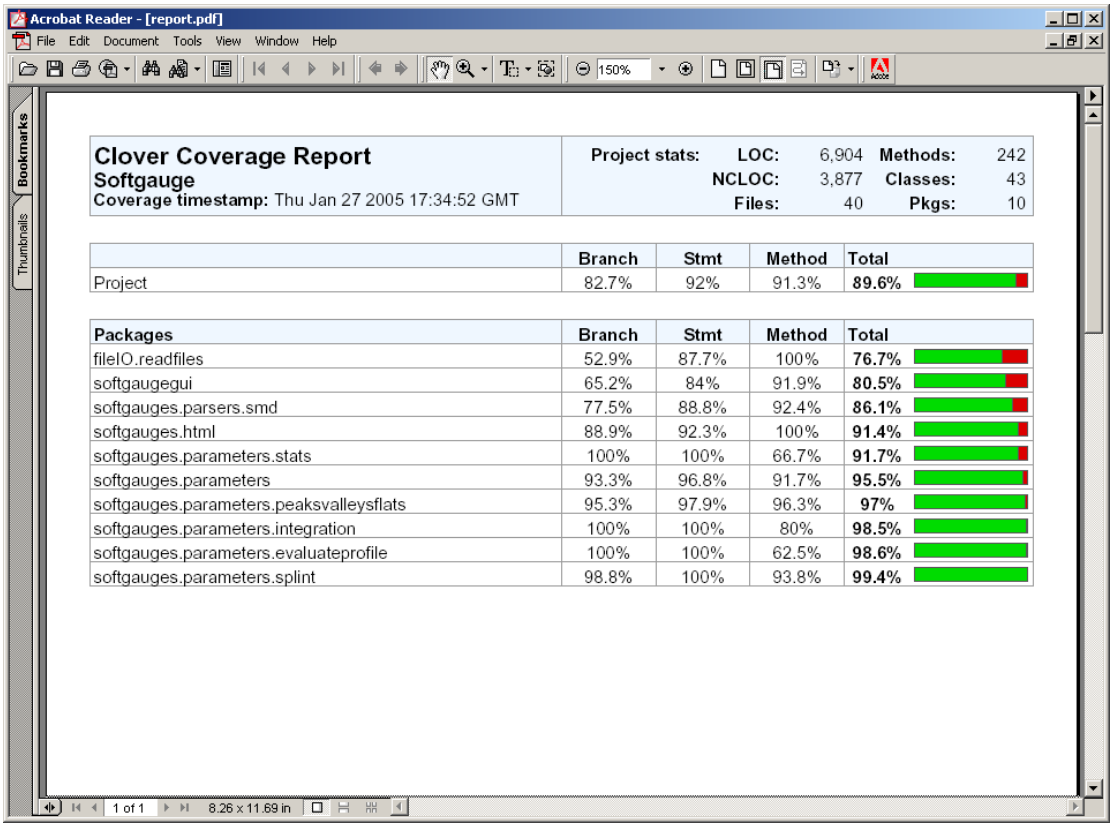

Figure 2: Screenshot of Clover Coverage Report

5.7 Delivery

This software was delivered as a *Jar* file. This is the standard way to package Java class files and resources that comprise an application, in a compressed form, together with version information.

To install the software, a zip file is downloaded from the web. It consists of:

`nplsmd.jar`	the software
`doc.zip`	the documentation for the software
`sin123withrefvalues.smd`	the SMD file
`sin123withrefvalues.html`	the HTML file
`softwarelicenceagreement.html`	the software end-user licence agreement

5.8 Maintenance

The web service *Bugzilla* [8] is used to record and track bug reports and fixes. This is an open-source bug tracking software.

6. Mapping onto NPL software quality procedure

NPL's quality procedure NPLQP/M13 is used at NPL by those developing software that must be fit-for-purpose and where the software process has to conform ISO 9000. The BPG1 assumes that a software process conforming to ISO 900 is in place. This section shows how the procedure was applied to the *softgauges* project.

The procedure makes the point that given the nature of the work an NPL it is not always possible to complete a functional requirement at the start of a development, so an iterative process of gathering user requirements, producing code and reviewing progress, until the functional requirements are complete. The resultant software may be used, or further development may commence using this as the basis of the functional requirement.

In the case of the *softgauges* project, the functional specification was produced using MATLAB as explained in the main body of this report.

6.1 Software integrity level

The quality requirements for software development depend on the use to which the software will be put and the consequences of its going wrong. These factors are quantified using an indicator called the "Software Integrity Level" (SIL). The SIL can be determined from a set of tables and a quality plan can then be produced from these. An application has been developed to do this based on completing some questions.

The SIL is determined from assessments of the criticality of usage and the complexity of the program. Procedure M13 provides tables that can be used to obtain numbers for these two factors.

6.1.1 Criticality of usage

Table 5 below shows the criticality of usage.

CU	Criticality of usage	Explanation
1	Not critical	• Has negligible effect on overall result • No danger of loss of income or reputation • Short life, will not require maintenance in future
2	Significant	• May result in incorrect results but will definitely be spotted (for example, by human review) • Potential for loss of income or reputation if faulty
3	Substantial	• May result in seriously incorrect result or errors may not be spotted (for example, by human review). • Likely to lead to loss of income or reputation if faulty
4	Life critical	• May result in personal injury or loss of life

Table 5: Criticality of Usage

The criticality of usage of the *softgauges* project is 3 (substantial).

6.1.2 Complexity of program

Table 6 below shows the complexity of the program.

CP	Complexity of processing	Typical features
1	Very simple	• Elementary functionality, easy to understand • Little or no control of an external system • Simple mathematics: for example, evaluation of a simple formula
2	Simple	• Simple functionality • Straightforward control of a system • Intermediate mathematics
3	Moderate	• Large or very large programs • Difficult to modify • Complicated mathematics
4	Complex	• Extremely complex functionality • Complex feedback systems • Very complicated mathematics

Table 6: Complexity of Program

The complexity of program of the *softgauges* project is 3 (moderate).

6.1.3 Overall SIL

The overall SIL is determined from a table provided in the procedure. This is shown as table 7 below, with the values for the *softgauges* project highlighted:

	CP1	CP2	CP3	CP4
CU1	1	1	1	1
CU2	2	2	3	4
CU3	3	3	3	4
CU4	4	4	4	4

Table 7: Overall Software Integrity Level

The table shows that *softgauges* has a SIL of 3.

The procedure provides a table of moderating factors that can increase or decease the SIL. Three factors, “Alternative means of verification”, “Modular approach”, and “Suitably trained staff available” were applicable to this project. Though they all have the potential to decrease the SIL, it was decided to leave the SIL unchanged at 3.

6.2 Quality Plan

The procedure provides tables (in an appendix) that show all the recommended and mandatory quality requirements for each SIL. These are used to draw up a quality plan. The quality plan for *softgauges* is shown in the table 8.

User requirements	Documented user requirement Review by team Review by suitably qualified independent person
Specification	Documented functional requirements Traceable requirements Review by team Review by suitably qualified independent person
Design	Clear and well structured design Tool support Review by team Review by suitably qualified independent person
Coding	Header to identify program name, author, date and version Program history Coding guidelines
Verification	Module testing as coding progresses Informal testing of complete software against specification
Validation	Documented testing against specification Review by team Review by suitably qualified independent person
Delivery, use and maintenance	Version control on release Version control before release Bug tracking / error logging Traceability User Documentation

Table 8: *Softgauges* Quality Plan

The following sections describe how each element of the quality plan was met. Each stage was reviewed by the team to see that the requirements were met.

The project to develop the softgauges software was part of a larger collaborative project involving length measurement scientists at NPL and other organizations. These collaborators provided the qualified independent review required at different stages of the software development process.

6.3 User Requirement

All the information for the user requirements of *softgauges* project was contained in the project proposal and agreed with the customer. The user requirements were therefore reviewed by the software team and by the collaborators on the wider *softgauges* project before the project started.

6.4 Functional Specification

For the *softgauges* project, the function specification covers three areas: parameters, SMD files and the GUI.

- The functional specification for parameters is written in MATLAB. The specification for the parts used as examples here appears in Appendix A of this document.
- A description of the SMD files is standardised and is given in [6].
- The GUI is only for demonstration purposes only.

  A brief specification of the GUI is the following:

  - Reads and displays the SMD files.
  - Calculates the parameters (as requested by parameters in Record 2).
  - Saves SMD files.
  - Allows saving of key information as an HTML file.

The (MATLAB) specification was reviewed by those responsible for the Java implementation, and the code design was reviewed by those responsible for the MATLAB specification.

6.5 Design

The design is captured in the IDE BlueJ used for development of the software and in the documentation of the code. The BlueJ diagram is shown in figure 1 (earlier in the report). The structure is derived from the code documentation. All the code has been documented using JavaDoc [13], which is a tool for generating API (Application Programmers Interface) documentation in HTML format from marked up comments in the source code.

Coding style is checked against agreed conventions stored in a file, using a tool called CheckStyle [2, 4], and any discrepancies are altered manually. The coding conventions used are mainly those recommended by SUN.

Tools used are:

- BlueJ 2.0.3 – IDE
- Java 1.4.2_4 – Java compiler etc.
- CheckStyle extension for BlueJ 3.3 – Code reviewer
- Ant apache-ant-1.6.2 – Equivalent to “Make”
- JUnit junit3.8.1 – unit testing
- Clover 1.3_02 – Code coverage

6.6 Coding

The quality issues are covered by comments included with the code. Coding style is as discussed in sections 5.5.1and 5.5.2.

6.7 Software Verification

The procedure requires that the software be tested to determine if it meets the requirements as developed so far on the required platform. If it does not, another cycle is repeated. The tests must be documented for any released version.

The testing that was done for *softgauges* is explained earlier in the report in section 5.6. The JUnit and Clover test results are stored in text files; an extract is shown in Figure 3.

6.8 Validation

This covers a review of the adequacy of both the requirements and the software, in order to ascertain if the output meets the overall aim specified in the User Requirements, and whether it delivers the functions listed in the Functional Requirements.

The testing and validation processes were presented to the wider *softgauges* project collaborators for their review and approval.

6.9 Delivery, Use and Maintenance

The version number and the location of each released version of the software need to be recorded. Pre-release versions need to be recorded in the same way. *Softgauges* versions are stored as sub-folders of a parent folder, together with a user-manual.

The release consists of the code in the form of a jar file, documentation, manual, and a sample (SMD) file with its HTML output.

Bugs are recorded using the web service *Bugzilla* [8].

```
Buildfile: build.xml …

with.clover:

[clover-setup] Clover Version 1.3.9, built on July 06 2005 …

code: …

    [javac] Compiling 50 source files to
F:\projects\ms_maths_and_stats\SoftGaugesPMH\LengthProgramme\softgaugesISO54
36-2-2001\source30092005Version1.01\build

   [clover] Clover Version 1.3.9, built on July 06 2005

   [clover] loaded from: C:\apache-ant-1.6.2\lib\clover.jar

   [clover] Workstation License registered to graeme.parkin@npl.co.uk,
Natiional Physical Laboratory

   [clover] Updating database at
'F:\projects\ms_maths_and_stats\SoftGaugesPMH\LengthProgramme\softgaugesISO5
436-2-2001\source30092005Version1.01\softgauge_coverage.db'

   [clover] Clover all over. Instrumented 40 files. …

BUILD SUCCESSFUL

Total time: 28 seconds

Buildfile: build.xml …

runtests:

    [junit] Testsuite: softgauges.parameters.SoftGaugesTest

    [junit] Tests run: 18, Failures: 0, Errors: 0, Time elapsed: 135.624 sec


    [junit] Testcase: testSoftGauge1 took 3.266 sec

    [junit] Testcase: testSoftGauge2 took 1.765 sec

…

    [junit] Testcase: testSoftGauge16 took 0.891 sec

…

    [junit] Testsuite: softgauges.parsers.smd.ByteInputTest

    [junit] Tests run: 17, Failures: 0, Errors: 0, Time elapsed: 0.297 sec


    [junit] Testcase: testreadByte took 0.141 sec

    [junit] Testcase: testreadBytesToString took 0.015 sec

…

    [junit] Testcase: testreadEndOfLine_2 took 0 sec

    [junit] Testcase: testreadEndOfLine_3 took 0.016 sec …

BUILD SUCCESSFUL

Total time: 5 minutes 8 seconds
```

Figure 3: Extracts from tests results

7. Conclusions

The development and testing of the *softgauges* software was successfully completed following the methodology of BPG1 and following the NPL software procedures in NPLQP/M013. The methodology of the BPG1 was sufficient to ensure that the *softgauges* software was fit-for-purpose and ensured that the software development process met the requirements of NPLQP/M013 and conformed to ISO 9000.

The BPG1 provides straightforward instructions for conducting an objective risk analysis of measurement software. This analysis results in a single number (the "Measurement Software Level") that is used to derive the validation techniques and tools that are required for the software. Basing the selection of techniques and tools on a single number makes it is quite clear how to tackle software development for measurement systems.

The techniques from the BPG1 must be applied during the software development process: these are not validation techniques to be applied to finished software. Most of the techniques are dependent on the use of tools, certainly for Measurement Software Level 2 (and above). These aspects of the application of Best Practice Guide No.1 should be compared with a previous case study [1], where the BPG1 was applied to a completed software development project, limited to using only a few general-purpose tools.

References

1. Barker, R. and G. Parkin, *Techniques for Validation of Measurement Software – without specialised tools.* NPL Report CMSC 43/04, NPL, March 2004. http://www.npl.co.uk/ssfm/download/documents/cmsc43_04.pdf

2. Burn, O., *Checkstyle 4.3*, 2005. http://checkstyle.sourceforge.net/

3. Cenqua, *Clover Code Coverage for Java*, 2007. http://www.cenqua.com/clover/

4. Giles, R., *BlueJ Checkstyle Extension Home Page*, 2007. http://bluejcheckstyle.sourceforge.net/

5. IEC 61508: 1998. *Functional safety: safety-related systems. Parts 1-7.*

6. ISO 5436-2: 2001. *Geometrical Product Specifications (GPS) – Surface texture: Profile method; Measurement standards – Part 2: Software measurement standards.*

7. Kölling, M., *BlueJ – The interactive Java environment*, 2007. http://www.bluej.org/

8. Mozilla, *Bugzilla*, 2007. http://www.bugzilla.org/

9. NPL, Taylor-Hobson, and CPT, *Softgauges*, 2006. http://161.112.232.32/softgauges/

10. Object Mentor, *JUnit, Testing Resources for Extreme Programming*, 2004. http://www.junit.org

11. Sun, *Java Language Specification – Expressions*, 2000. http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html

12. Sun, *Code Conventions for the Java Programming Language*, 2007. http://java.sun.com/docs/codeconv/

13. Sun, *JavaDoc Tool*, 2007. http://java.sun.com/j2se/javadoc/

14. Sun, *The Source for Java Developers*, 2007. http://java.sun.com

15. The Mathworks, *MATLAB - The Language of Technical Computing*, 2007. http://www.mathworks.com/products/matlab/

16. 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/documents/ssfmbpg1.pdf

Appendix A MATLAB Specification

A.1 The main routine: SoftGauge

Main routine for evaluation of surface texture parameters.

```
  function [ ParamVal, z, x, z2, SampInd ] = ...
      SoftGauge(ParamStr, p, DeltaX, LambdaC)
% ------------------------------------------------------------
% SoftGauge
%
% Author        Peter Harris
% Created       26-05-2004
% Amended       11-10-2004
% ------------------------------------------------------------
% Evaluate surface texture parameter.
%
% Input
%    ParamStr   parameter (string)
%    p          primary profile ordinate values
%    DeltaX     increment in profile abscissa values
%    LambdaC    filter cut-off (only for 'R' and 'W')
%
% Output
%    ParamVal   surface texture parameter values:
%                  aggregated Pa, Ra or Wa value
%                  aggregated Pq, Rq or Wq value
%                  aggregated Psk, Rsk or Wsk value
%                  aggregated Pku, Rku or Wku value
%                  aggregated Pp, Rp or Wp value
%                  aggregated Pv, Rv or Wv value
%                  aggregated Pz, Rz or Wz value
%                  aggregated PSm, RSm or WSm value
%                  aggregated Pc, Rc or Wc value
%                  Pt, Rt or Wt value
%    z          profile ordinate values
%    x          profile abscissa values
%    z2         second derivatives of interpolating function
%                  at profile abscissa values
%    SampInd    indices of endpoints defining sample lengths
%
% Notes
%    1. Assume form has been removed.
%    2. Assume uniformly-spaced abscissa values.
% ------------------------------------------------------------

% ------------------------------------------------------------
% Evaluate profile to be analysed:
%    z          profile ordinate values
%    x          profile abscissa values
%    SampInd    indices of endpoints defining sample lengths
%    CN         calculation number
%    SampLen    sample length
%    EvalLen    evaluation length
%
  [ z, x, SampInd, CN, SampLen, EvalLen ] = ...
      EvaluateProfile(ParamStr, p, DeltaX, LambdaC);
%
```

```
% ------------------------------------------------------------
% Natural cubic spline interpolant to the profile points
% (augmented with zero-crossing points and sample length
% endpoints).
%
  [ z2 ] = NatSplInt(z, x);
%
% ------------------------------------------------------------
% For each sample length ...
%
  for k = 1 : CN
%
%    Evaluate surface texture parameters in terms of
%    interpolant.
%
     ParamRa(k)  = EvalParamRa( z, x, z2, SampInd, k, SampLen);
     ParamRq(k)  = EvalParamRq( z, x, z2, SampInd, k, SampLen);
     ParamRsk(k) = EvalParamRsk(z, x, z2, SampInd, k, SampLen);
     ParamRku(k) = EvalParamRku(z, x, z2, SampInd, k, SampLen);
     ParamRp(k)  = EvalParamRp( z, x, z2, SampInd, k, SampLen);
     ParamRv(k)  = EvalParamRv( z, x, z2, SampInd, k, SampLen);
     ParamRz(k)  = EvalParamRz( z, x, z2, SampInd, k, SampLen);
     ParamRSm(k) = EvalParamRSm(z, x, z2, SampInd, k, SampLen);
     ParamRc(k)  = EvalParamRc( z, x, z2, SampInd, k, SampLen);
%
  end
%
% Evaluate aggregated surface texture parameters.
%
  ParamVal(1) = mean(ParamRa);
  ParamVal(2) = mean(ParamRq);
  ParamVal(3) = mean(ParamRsk);
  ParamVal(4) = mean(ParamRku);
  ParamVal(5) = mean(ParamRp);
  ParamVal(6) = mean(ParamRv);
  ParamVal(7) = mean(ParamRz);
  ParamVal(8) = mean(ParamRSm);
  ParamVal(9) = mean(ParamRc);
%
% Evaluate surface texture parameter defined over evaluation
% length.
%
  ParamVal(10) = EvalParamRt0( z, x, z2, SampInd, CN, SampLen);
%
% ------------------------------------------------------------
% End.
```

A.2 EvalParamRa

Evaluation of parameters *Pa*, *Ra* and *Wa*.

```
  function [ ParamRa ] = EvalParamRa(z, x, z2, SampInd, k, SampLen)
% ------------------------------------------------------------
% EvalParamRa
%
% Author        Peter Harris
% Created       04-06-2004
% Amended       16-06-2004
% ------------------------------------------------------------
% Evaluation of parameters Ra, Wa, Pa.
%
% Input
%    z          profile ordinate values
%    x          profile abscissa values
%    z2         second derivatives of interpolating function
%                  at profile abscissa values
%    SampInd    indices of endpoints defining sample lengths
%    k          index of sample length for evaluation
%    SampLen    sample length
%
% Output
%    ParamRa    value of parameter Ra, Wa, Pa
% ------------------------------------------------------------

% ------------------------------------------------------------
% Accumulate estimates of integrations over each interval
% making up the sample langth, each estimate obtained by
% Romberg integration of order 4 (Simpson's rule).
%
  sum = 0;
  for ival = SampInd(k) : SampInd(k+1)-1
     ss  = RombInt('FuncRa', z, x, z2, ival, 2);
     sum = sum + ss;
  end
%
% ------------------------------------------------------------
% Evaluate parameter.
%
  ParamRa = sum/SampLen;
%
% ------------------------------------------------------------
% End.
```

A.3 RombInt

Function that does integration.

```
  function [ ss ] = RombInt(func, z, x, z2, ival, K)
% ------------------------------------------------------------
% RombInt
%
% Author        Peter Harris
% Created       04-06-2004
% Amended       23-06-2004
% ------------------------------------------------------------
% Implements Romberg's method of order 2K for the
% integration of a function.
%
% Input
%    func       name of function
%    z          profile ordinate values
%    x          profile abscissa values
%    z2         second derivatives of interpolating function
%                  at profile abscissa values
%    ival       index of interval of profile for integration
%
%    K          determines order of integration rule
%
% Output
%    ss         estimate of integration of function
%
% Notes
%    1. See "Numerical Recipes" function "qromb".
% ------------------------------------------------------------

% ------------------------------------------------------------
% Apply extended trapezoidal quadrature rule, and store
% relative step sizes.
%
  s(1) = 0;
  h(1) = 1;
  for j = 1 : K
     s(j)   = Trapzd(j, func, z, x, z2, ival, s(j));
     s(j+1) = s(j);
     h(j+1) = h(j)/4;
  end
%
% Apply polynomial extrapolation to achieve order 2K
% integration rule.
%
  ss = PolExt(h(1:K), s(1:K), K, 0);
%
% ------------------------------------------------------------
% End.
```

A.4 FuncRa

Function evaluation of parameters *Ra*, *Wa* and *Pa*.

```
  function [ fRa ] = FuncRa(z, x, z2, xval, ival)
% ------------------------------------------------------------
% FuncRa
%
% Author        Peter Harris
% Created       26-05-2004
% Amended       16-06-2004
% ------------------------------------------------------------
% Function evaluation for parameters Ra, Wa, Pa.
%
% Input
%    z          profile ordinate values
%    x          profile abscissa values
%    z2         second derivatives of interpolating function
%                  at profile abscissa values
%    xval       abscissa values for evaluation
%    ival       index of interval containing xval
%
% Output
%    fRa        evaluated function values
% ------------------------------------------------------------

% ------------------------------------------------------------
% Spline interpolation.
%
  zval = EvalSplInt(z, x, z2, xval, ival);
%
% ------------------------------------------------------------
% Function value.
%
  fRa = abs(zval);
%
% ------------------------------------------------------------
% End.
```

A.5 EvalSplInt

Function to evaluate spline interpolant.

```
  function [ zval ] = EvalSplInt(z, x, z2, xval, ival)
% ------------------------------------------------------------
% EvalSplInt
%
% Author        Peter Harris
% Created       26-05-2004
% Amended       04-06-2004
% ------------------------------------------------------------
% Evaluate spline interpolatant.
%
% Input
%    z          profile ordinate values
%    x          profile abscissa values
%    z2         second derivatives of interpolating function
%                  at profile abscissa values
%    xval       abscissa values for evaluation
%    ival       index of interval containing xval
%
% Output
%    zval       evaluated ordinate values
%
% Notes
%    1. See "Numerical Recipes" function "splint".
% ------------------------------------------------------------

% ------------------------------------------------------------
% Spline interpolation scheme.
%
  h = x(ival+1) - x(ival);
  a = (x(ival+1) - xval)/h;
  b = (xval - x(ival))/h;
  c = (a.^3 - a)*(h^2)/6;
  d = (b.^3 - b)*(h^2)/6;
  zval = a*z(ival) + b*z(ival+1) + c*z2(ival) + d*z2(ival+1);
%
% ------------------------------------------------------------
% End.
```

Appendix B Java implementation

Note that the comments conform to a set style in order to facilitate the automatic generation of documentation.

B.1 Package softgauges.parameters

Contains method `softGauge` to evaluate all the softgauges parameters.

```
package softgauges.parameters;

import java.util.regex.Pattern;
import softgauges.parameters.evaluateprofile.EvaluateProfileResults;
import softgauges.parameters.evaluateprofile.EvaluateProfile;
import softgauges.parameters.evaluateprofile.Filters;
import softgauges.parameters.splint.SplInt;

/**
 * Main routine for evaluation of surface texture parameters.
 *
 * @author Graeme I Parkin
 * @version     1.3 - 5 November 2004,    Added checks to stop wrong
 *                                        evaluation of parameters.
 * @version <BR>1.2 - 3 November 2004,    Revised to meet coding standards.
 * @version <BR>1.1 - 13 July 2004        Added new parameters:p, v, z, t
 * @version <BR>1.0 - 16 June 2004
 */
public strictfp class SoftGauges {
     /**
      *
      */
     protected SoftGauges() {
     }
     /**
      * Pattern for all parameters.
      */
     public static final String REALLP =
      "([P]|[W]|[R])(a|q|(ku)|(sk)|p|v|z|t|(Sm)|c)";
     /**
      * Pattern for [P|R|W]a.
      */
     public static final String REPRWA = "([P]|[W]|[R])a";
     /**
      * Pattern for [P|R|W]q.
      */
     public static final String REPRWQ = "([P]|[W]|[R])q";
     /**
      * Pattern for [P|R|W]ku.
      */
     public static final String REPRWKU = "([P]|[W]|[R])ku";
     /**
      * Pattern for [P|R|W]sk.
      */
     public static final String REPRWSK = "([P]|[W]|[R])sk";
     /**
      * Pattern for [P|R|W]p.
      */
     public static final String REPRWP = "([P]|[W]|[R])p";
     /**
      * Pattern for [P|R|W]v.
      */
     public static final String REPRWV = "([P]|[W]|[R])v";
```

```
/**
 * Pattern for [P|R|W]z.
 */
public static final String REPRWZ = "([P]|[W]|[R])z";
/**
 * Pattern for [P|R|W]t.
 */
public static final String REPRWT = "([P]|[W]|[R])t";
/**
 * Pattern for [P|R|W]Sm.
 */
public static final String REPRWSM = "([P]|[W]|[R])Sm";
/**
 * Pattern for [P|R|W]c.
 */
public static final String REPRWC = "([P]|[W]|[R])c";


/**
 * Evaluate surface texture parameter.
 *
 * @param  paramStr   parameter (defined by a string)
 * @param  x          primary profile abscissa values
 * @param  p          primary profile ordinate values
 * @param  deltaX     increment in profile abscissa values
 * @param  lambdaC    filter cut-off (only for 'R' and 'W')
 * @return z          - profile ordinate values,
 *         x          - profile abscissa values,
 *         ZeroInd    - indices of all zero crossing points,
 *         EndInd     - indices of endpoints defining sample lengths,
 *         ParameterValue - calculated parameter value
 * @throws Exception Captures exceptions
 */
public static SoftGaugeResults softGauge( final String paramStr,
                                          final double[] x,
                                          final double[] p,
                                          final double deltaX,
                                          final double lambdaC)
                                    throws Exception {
     // Evaluate profile to be analysed
     EvaluateProfileResults epr =
     EvaluateProfile.evaluateProfile(paramStr, x, p, deltaX, lambdaC);
     // Check that each sample length has a least one zero point.
     if (!onecrossingpointpersample(epr)) {
          throw new
               Exception("Missing crossing point in sample lengths");
     }
     // Natural cubic spline interpolant to the profile points
     // (augmented with zero-crossing points and sample length
     // endpoints).
     double[] z2 = SplInt.natSplInt(epr.z, epr.x);
     // Get function to evaluate the parameter
     SoftGaugeFunction sgf = getFunctionToEvaluateParameter(paramStr);
     // Evaluate the parameter
     SoftGaugeResults sgr = sgf.evalParam(epr, z2);
     return sgr;
}
```

```
/**
 * Select function to evaluate parameter.
 *
 * @param  paramStr   parameter (defined by a string)
 * @return Function   to evaluate the parameter
 * @throws Exception  invalid parameter type
 */
private static  SoftGaugeFunction
                getFunctionToEvaluateParameter(final String paramStr)
                throws Exception {
     SoftGaugeFunction sgf = null;
     if (Pattern.matches(REPRWA, paramStr)) {
          sgf = new EvalParama();
     }
     if (Pattern.matches(REPRWQ, paramStr)) {
          sgf = new EvalParamq();
     }
     if (Pattern.matches(REPRWKU, paramStr)) {
          sgf = new EvalParamku();
     }
     if (Pattern.matches(REPRWSK, paramStr)) {
          sgf = new EvalParamsk();
     }
     if (Pattern.matches(REPRWP, paramStr)) {
          sgf = new EvalParamp();
     }
     if (Pattern.matches(REPRWV, paramStr)) {
          sgf = new EvalParamv();
     }
     if (Pattern.matches(REPRWZ, paramStr)) {
          sgf = new EvalParamz();
     }
     if (Pattern.matches(REPRWT, paramStr)) {
          sgf = new EvalParamt();
     }
     if (Pattern.matches(REPRWSM, paramStr)) {
          sgf = new EvalParamSm();
     }
     if (Pattern.matches(REPRWC, paramStr)) {
          sgf = new EvalParamc();
     }
     // Check parameter found
     if (sgf == null) {
          throw new Exception("Invalid parameter type: >" +
                              paramStr + "<");
     }
     return sgf;
}
```

```
/**
 * Calculate the units of parameters.
 * @param  paramStr   parameter (defined by a string)
 * @param  unitz      units of z
 * @param  unitx      units of x
 * @return            unit of parameter
 * @throws Exception  if not valid parameter
 */
public static String parameterUnit ( final String paramStr,
                                     final String unitz,
                                     final String unitx)
                                throws Exception {
     String unit = "";
     if (Pattern.matches(REPRWA, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWQ, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWKU, paramStr)) {
          unit = "1";
     }
     if (Pattern.matches(REPRWSK, paramStr)) {
          unit = "1";
     }
     if (Pattern.matches(REPRWP, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWV, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWZ, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWT, paramStr)) {
          unit = unitz;
     }
     if (Pattern.matches(REPRWSM, paramStr)) {
          unit = unitx;
     }
     if (Pattern.matches(REPRWC, paramStr)) {
          unit = unitz;
     }
     // Check parameter found
     if (unit.equals("")) {
          throw new Exception("Invalid parameter type: >" +
                              paramStr + "<");
     }
     return unit;
}
```

```
/**
 * Checks parameters required by the softgauges functions and
 * forces abscissa to be equally spaced.<BR>
 *   1) No of abscissa values >=2, increasing in value (xip1 > xi).<BR>
 *   2) Sufficient data to filter.<BR>
 *
 * @param  paramStr   parameter (defined by a string)
 * @param  x          primary profile abscissa values modified to make
 *                    equally spaced
 * @param  p          primary profile ordinate values
 * @param  deltaX     increment in profile abscissa values
 * @param  lambdaC    filter cut-off (only for 'R' and 'W')
 * @return true if OK
 * @throws Exception  Captures exceptions
 */
public static boolean checkParameters(    final String paramStr,
                                          double[] x,
                                          final double[] p,
                                          final double deltaX,
                                          final double lambdaC)
                                    throws Exception {
     boolean check = true;
     //
     // Condition 1
     if ((x.length < 2) || !increasing(x)) {
          check = false;
     } else {
          // If parameter of type P then complete
          if (paramStr.startsWith(EvaluateProfile.PPROFILE)) {
               check = true;
          } else {
               // Condition 2 for parameters R and W
               int n = p.length;
               int m = (int) (lambdaC / deltaX);
               double l = (n - 2 * m - 1) * deltaX;
               if ((l < lambdaC) || (m < Filters.MINDATAGAUSSIAN)) {
                    check = false;
               } else {
                    if (  (paramStr.
                                startsWith(EvaluateProfile.RPROFILE))
                       || (paramStr.
                                startsWith(EvaluateProfile.WPROFILE))) {
                         check = true;
                    } else {
                         throw new Exception("Invalid parameter: " +
                                             paramStr);
                    }
               }
          }
     }
     // Now force the abscissa to be equally spaced.
     if (check) {
          double space = (x[x.length - 1] - x[0]) / (x.length - 1);
          for (int index = 1; index < x.length; index++) {
               x[index] = x[0] + space * index;
          }
     }
     return check;
}
```

```
/**
 * Check all values in array are increasing.
 *
 * @param x array of values
 * @return true if all values increasing
 */
public static boolean increasing(final double[] x) {
     boolean check = true;
     for (int index = 0; index < (x.length - 1); index++) {
          if (x[index] >= x[index + 1]) {
               check = false;
               break;
          }
     }
     return check;
}

/**
 * Checks for at least one crossing point in each sample length.
 * @param epr contains data for the softgauge function
 * @return true if at least one crossing point per sample length
 */
public static final boolean onecrossingpointpersample(
                              final EvaluateProfileResults epr) {
     boolean check = true;
     for (int index = 1; index <= epr.cn; index++) {
          if (!crossingpoint(epr.z, epr.endInd[index - 1],
                                    epr.endInd[index])) {
               check = false;
               break;
          }
     }
     return check;
}

/**
 * Gets the min value of the array.
 *
 * @param x array of values
 * @param index1 index to start of array to evaluate
 * @param index2 index to end of array to evaluate
 * @return min value of array
 */
public static final double min( final double[] x,
                              final int index1,
                              final int index2) {
     double minvalue = x[0];
     for (int index = index1; index <= index2; index++) {
          if (x[index] < minvalue) {
               minvalue = x[index];
          }
     }
     return minvalue;
}
```

```
     /**
      * Gets the max value of the array.
      *
      * @param x array of values
      * @param index1 index to start of array to evaluate
      * @param index2 index to end of array to evaluate
      * @return max value of array
      */
     public static final double max( final double[] x,
                                     final int index1,
                                     final int index2) {
          double maxvalue = x[0];
          for (int index = index1; index <= index2; index++) {
               if (x[index] > maxvalue) {
                    maxvalue = x[index];
               }
          }
          return maxvalue;
     }

     /**
      * Determines if crossing points exists.
      *
      * @param x array of values
      * @param index1 index to start of array to evaluate
      * @param index2 index to end of array to evaluate
      * @return true if at least one crossing point
      */
     public static final boolean crossingpoint(     final double[] x,
                                                    final int index1,
                                                    final int index2) {
          return max(x, index1, index2) > 0 && min(x, index1, index2) < 0;
     }
}
```

B.2 EvalParama

`EvalParama` contains the method `evalParam` to evaluate the parameters *Pa*, *Ra* and *Wa*.

```
package softgauges.parameters;

import softgauges.parameters.evaluateprofile.EvaluateProfileResults;
import softgauges.parameters.stats.Stats;
import softgauges.parameters.splint.SplInt;
import softgauges.parameters.integration.Integration;

/**
 * Evaluation of parameters Pa, Ra and Wa.
 *
 * @author Graeme I Parkin
 * @version     1.1 - 3 November 2004     Revised to meet coding standards.
 * @version     1.0 - 16 June 2004
 */
public strictfp class EvalParama implements SoftGaugeFunction {
     /**
      * Evaluation of parameters Pa, Ra and Wa
      * (Revised to include calculation of mean).
      *
      * @param  epr        evaluation profile results
      * @param  z2         second derivatives of interpolating function
      *                    at profile abscissa values
      * @return ParamRa    - Evaluation of parameter value
      * @throws Exception  Catches all exceptions
      */
     public final SoftGaugeResults
                    evalParam( final EvaluateProfileResults epr,
                               final double[] z2)
                          throws Exception {
          SoftGaugeResults sgr = new SoftGaugeResults();
          sgr.z = epr.z;
          sgr.x = epr.x;
          sgr.zeroInd = epr.zeroInd;
          sgr.endInd = epr.endInd;
          sgr.z2 = z2;
          // For each sample length ...
          double[] param = new double [epr.cn];
          // Get function to evaluate the parameter
          for (int k = 1; k <= epr.cn; k++) {
          // Evaluate surface texture parameter in terms of interpolant.
               param[k - 1] = evalParamk(epr.z, epr.x, z2, epr.zeroInd,
                                         epr.endInd, k, epr.sampLen);
          }
          // Evaluate aggregated surface texture parameter.
          double param0 = Stats.mean(param);
          sgr.parameterValue = param0;
          return sgr;
     }
```

```
    /**
     * Evaluation of parameters Pa, Ra and Wa.
     *
     * @param  z          profile ordinate values
     * @param  x          profile abscissa values
     * @param  z2         second derivatives of interpolating
     *                    function at profile abscissa values
     * @param  zeroInd    indices of zero crossing points
     * @param  endInd     indices of endpoints defining sample lengths
     * @param  k          index of sample length for evaluation
     * @param  sampLen    sample length
     * @return ParamRa    - Evaluation of parameter value
     */
    public final double evalParamk( final double[] z,
                                    final double[] x,
                                    final double[] z2,
                                    final int[] zeroInd,
                                    final int[] endInd,
                                    final int k,
                                    final double sampLen) {
        double sum = 0;
        for (int ival = endInd[k - 1]; ival < endInd[k]; ival++) {
            double ss = Integration.rombInt(new EvalParama(),
                                            z, x, z2, ival, 2);
            sum = sum + ss;
        }
        return sum / sampLen;
    }

    /**
     * Softgauge function parameter evaluation.
     *
     * @param  z    profile ordinate values
     * @param  x    profile abscissa values
     * @param  z2   second derivatives of interpolating
     *              function at profile abscissa values
     * @param  xval abscissa value for evaluation
     * @param  ival index of interval containing xval
     * @return fRa - Evaluation of parameter value
     */
    public final double parameterFunction(    final double[] z,
                                              final double[] x,
                                              final double[] z2,
                                              final double xval,
                                              final int ival) {
        double zval = SplInt.evalSplInt(z, x, z2, xval, ival);
        return StrictMath.abs(zval);
    }
}
```

B.3 rombInt

`RombInt` for integration of a function is implemented in a package called `softgauges.parameters.integration`, some of which is shown below.

```
/**
 * Implements Romberg's method of order 2K for the
 * integration of a function.
 *
 * @param func function to be integrated
 * @param  z    profile ordinate values
 * @param  x    profile abscissa values
 * @param  z2   second derivatives of interpolating function at profile
 *              abscissa values
 * @param  ival index of profile for integration
 * @param  k    determines order of integration rule
 * @return      the estimate of the integration of the function
 */
public static double rombInt(   final SoftGaugeFunction func,
                                final double[] z,
                                final double[] x,
                                final double[] z2,
                                final int ival,
                                final int k) {
     double[] s = new double[k];
     double[] h = new double[k];
     s = trapzd(k, func, z, x, z2, ival);
     h[0] = 1;
     for (int j = 1; j < k; j++) {
          h[j] = h[j - 1] / 4;
     }
     double ss = polExt(h, s, k, 0);
     return ss;
}
```

```
/**
 * Implement the first n stages of refinement of extended trapezoidal
 * rule for the integration of a function.
 *
 * @param  n    number of refinements
 * @param  func function to be integrated
 * @param  z    profile ordinate values
 * @param  x    profile abscissa values
 * @param  z2   second derivatives of interpolating function at profile
 *              abscissa values
 * @param  ival index of profile for integration
 * @return      the estimates of the integration of the function
 */
public static double[] trapzd(  final int n,
                                final SoftGaugeFunction func,
                                final double[] z,
                                final double[] x,
                                final double[] z2,
                                final int ival) {
     double a = x[ival];
     double b = x[ival + 1];
     double h = b - a;
     double[] s = new double[n];
     double fa = func.parameterFunction(z, x, z2, a, ival);
     double fb = func.parameterFunction(z, x, z2, b, ival);
     s[0] = (fa + fb) * h / 2;
     for (int index = 1; index < n; index++) {
          int np = (int) (pow(2, index - 1)); //(Math.pow(2, n - 2));
          double hp = h / np;
          double xi = a + hp / 2;
          double sum = 0;
          for (int j = 0; j < np; j++) {
               double fxi = func.parameterFunction(z, x, z2, xi, ival);
               sum = sum + fxi;
               xi = xi + hp;
          }
          s[index] = (s[index - 1] + h * sum / np) / 2;
     }
     return s;
}
```

B.4 evalSplInt

The MATLAB function `FuncRa` is implemented in Java by the method `EvalParama.parameterFunction`, which calls `evalSplInt`, in a package called `softgauges.parameters.splint`; `evalSplInt` is an implementation of the MATLAB function `EvalSplInt`.

```
/**
 * Evaluate natural cubic spline interpolant.
 *
 * @param  z    profile ordinate values
 * @param  x    profile abscissa values
 * @param  z2   second derivatives of interpolating function at
 *              profile abscissa values
 * @param  xval abscissa value for evaluation
 * @param  ival index of interval containing xval
 * @return zval - evaluated ordinate value
 */
public static double evalSplInt(     final double[] z,
                                     final double[] x,
                                     final double[] z2,
                                     final double xval,
                                     final int ival) {
     double h = x[ival + 1] - x[ival];
     double a = (x[ival + 1] - xval) / h;
     double b = (xval - x[ival]) / h;
     double c = (a * a * a - a) * h * h / 6;
     double d = (b * b * b - b) * h * h / 6;
     double zval =   a *  z[ival] + b * z[ival + 1] +
                     c * z2[ival] + d * z2[ival + 1];
     return zval;
}
```

B.5 runPRWforSoftGauge

The routine `runPRWforSoftGauge` is used to run tests for back to back testing with the MATLAB specifications. The data from the MATLAB is saved to files. This is used by the JUnit tests like `testSoftGauge1` for parameters *Pa*, *Ra* and *Wa*, which calls `runPRWforSoftGauges`, which in turn calls `runPRWforSoftGauge`.

```
/**
 * Runs SoftGauge with different data.
 * @param  stestdirectory       directory containing inputs and results
 * @param  sParamName           parameter name
 * @param  pFile                File name for p data
 * @param  deltaXLambdaCFile    file name for DeltaX and LambdaX data
 * @param  parametersFile       File name containing the parameters data
 * @param  columnofresults      column with results
 * @param  tolerance            degree of accuracy
 * @throws Exception            Catches any exceptions
 */
public final void runPRWforSoftGauge(final String stestdirectory,
                                     final String sParamName,
                                     final String pFile,
                                     final String deltaXLambdaCFile,
                                     final String parametersFile,
                                     final int columnofresults,
                                     final double tolerance
                                )
                                throws Exception {
     String testdirectory = stestdirectory;
     // Set up parameters for the call
     String paramName = sParamName;
     String filename = testdirectory + pFile;
     double[][] results =
               fileIO.readfiles.filereaders.readcolumns(filename, 1);
     double[] p = results[0];
     filename = testdirectory + deltaXLambdaCFile;
     results = fileIO.readfiles.filereaders.readdatalines(filename);
     double deltaX = results[0][0];
     double lambdaC = results[1][0];
     // Create X data
     double[] x = new double[p.length];
     for (int index = 0; index < x.length; index++) {
          x[index] = index * deltaX;
     }
     // Call SoftGauge
     SoftGaugeResults spr =
          SoftGauges.softGauge(paramName, x, p, deltaX, lambdaC);
     // Check the results
     filename = testdirectory + parametersFile;
     results = fileIO.readfiles.filereaders.readcolumns(filename, 8);
     double[] parameter = results[columnofresults];
     assertEquals(   parameter[parameter.length - 1],
                     spr.parameterValue,
                     tolerance);
}
```

```
/**
 * SoftGauge <BR>
 * Test:   1 <BR>
 * Aim:    Test SoftGauge for [P|R|W]a.
 */
public final void testSoftGauge1() {
     try {
          runPRWforSoftGauges("a", 0, TOLERANCEA);
     } catch (Exception e) {
          fail("No exception should occur: " + e.getMessage());
     }
}
```