inspearit
Aligning for Customer Value

The « SQALE » Models for assessing the Quality of Software Source Code

Measuring the source code quality of a software application is a significant advantage.
The "SQALE" (Software Quality Assessment based on Lifecycle Expectations) method developed by inspearit makes it possible to:
- assess the application's quality to provide visibility,
- provide quantitative diagnosis for all software types and suppliers,
- organise the findings and clearly proritize the remediation action plan,
- be easily and quickly implemented,
- complement process assessment methods.

The "SQALE" method presented here entails the use of two original models - a ranked and uniform quality model and an analysis model based on remediation costs.
The results are objectively comparable between applications.

Quality Model and Analysis Model

The need to assess and know the quality of the software one has required or payed for is not new.

When an application is accepted, in addition to functional and performance testing, it should be possible to obtain easily and precisely other quality attributes of the expected deliverable.

For example, it should be possible to know how the code will accept future evolutions (which are bound to come), or whether it will be easily possible to outsource its maintenance to a third party team.

Although some large customers have developed their own methods, or even their own tools to satisfy such assessment needs, regularly assessing the delivered software has not been generalized. This situation is partly the result of a lack of an accepted standardized method allowing systematic assessments to be performed within reduced schedules and costs.

This absence of methods to assess software is also felt by software development managers. They lack standard and simple means to assess the quality of developed software in order to be able to monitor and control their projects within the three major axes that are quality, cost and schedule.

A software development organization's maturity level and the capabilities of its processes may be assessed and measured on a scale as provided by a standard model such as CMMI.

However, as far as the software product resulting from these processes is concerned, there is no complete and calibrated system to assess and rate its quality.

Best practice models such as CMMI for Acquisition[1] or CMMI for Development[2] do require the control and the monitoring of the quality of the products during the whole project lifecycle.

The CMMI de facto standard may be used to appraise *processes* implemented in an organization. However, to appraise the software *product*, a standard is still needed in practice.

For example, this is the case of the practice SP 1.2 from the Process Area (PA) Product and Process Quality Assurance (PPQA) "Objectively Evaluate Work Products and Services" and the practice SP 2.3 of the PA Supplier Agreement Management (SAM) "Evaluate Selected Supplier Work Products".

Such practices would be much easier to implement if a simple and standard method was available to rate software source code.

The two main components of an assessment method are on the one hand, a quality model, and on the other hand, an analysis model.

- The quality model defines the quality characteristics (or attributes), expected in the context of the assessment. Characteristics are broken down into detailed sub-characteristics. Such a breakdown allows the definition of precise controls to be set up for the product assessment.

- The analysis model defines the rules to characterise or rate the products (or components thereof) based on measurements obtained on the control points of the quality model. This is the "analysis model" defined in standard ISO/IEC 15939[3].

The best known quality models are from Boehm[4], McCall[5] or ISO 9126[6]. Such models are general and have not been specifically developed to rate the source code of an application.

Several issues arise when trying to apply them to the assessment of software source code:

- Functional and external characteristics are mixed together with internal characteristics.

- The models present a different viewpoint from the assessor's point of view who is responsible for rating the quality of a source code.

- The defined characteristics of the model are non-orthogonal, sharing redundant properties.

But the great weakness of such models is that none provide a practical and concrete method to rate an application.

In practice, in order to apply such models one must create one's own assessment method. For example, a panel of experts may be gathered to give a rating for each sub-characteristic, between 1 and 5. Averaging the ratings, it will be possible to identify the "good" sub-characteristics and the "weak" ones. Of course, such a method is costly, takes time to implement, and is not repeatable.

Therefore DNV has developed the SQALE method and models to satisfy the needs in assessing strategic large scale software source code. We have been successfully using this method for several years for our customers.

We describe hereafter the two SQALE models to contribute to the collective work necessary to build a common assessment standard.

The SQALE Quality Model

This model has four important properties:

- It has been built in a systematic manner taking into account the lifecycle of source code.

- It is structured in ranked layers.

- It is a requirement model and not a set of best practices to implement.

- It takes into account both the developer's point of view and the software user's point of view.

To define the basic quality characteristics of our model, an approach based on the typical lifecycle of a source code file was used (Figure 1).

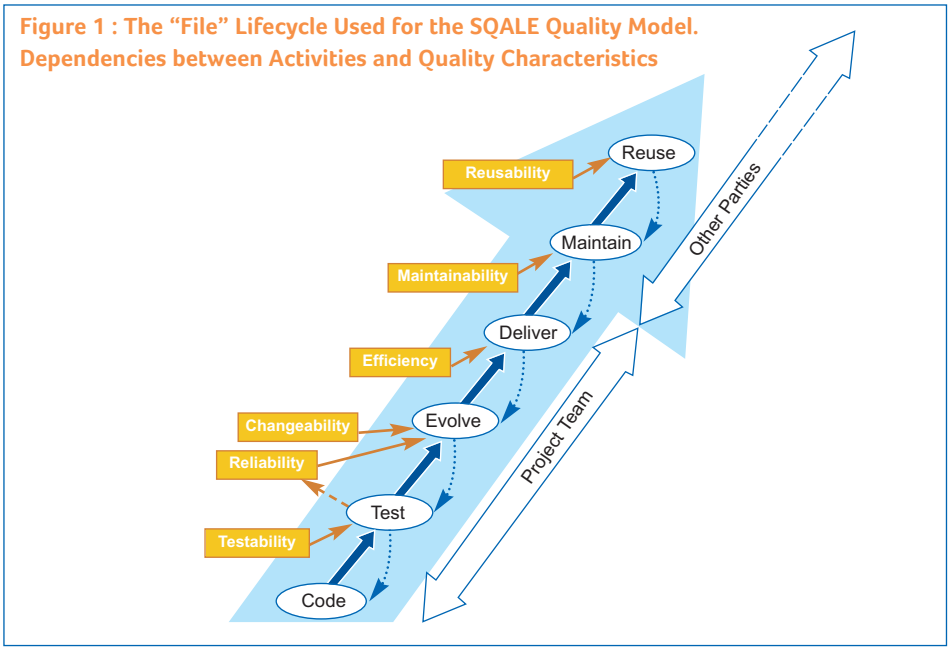
This file granularity level was retained since within development project, the common practice is that "owners" are usually assigned to files.

It is also the granularity that is commonly supported by configuration management tools, compilers and linkers.

Whichever the overall lifecycle (V life cycle, spiral or iterative lifecycle) of an application, a file will follow a sequence similar to the one described in figure 1.

Even if there are on the overall level comings and goings between different phases (for example between coding and testing), Figure 1 illustrates a general time

Figure 1 : The "File" Lifecycle Used for the SQALE Quality Model. Dependencies between Activities and Quality Characteristics



sequence and some practical rules:

- An untested code will not be delivered.
- Evolutions must be taken into account and subsequent corrections made before delivery.
- An unmaintainable code will not be re-used.

We asked ourselves what the major quality prerequisites are for each step in the life cycle? We used the quality characteristics and sub-characteristics of the 3 models (Boehm[4], McCall[5] and ISO 9126[6]) which allowed us to establish the dependencies presented in figure 1.

The needed "reusability" characteristic has been added as it is not included in the ISO 9126 model.

This is how this approach organizes sequentially the requirements and needs for source code quality. In this sequence, testability is the first characteristic and reusability is the last one required.

The layered model presented in figure 2 is the end result of this approach. This basic model may be adapted by taking into account the expected lifecycle for the source code to be assessed.

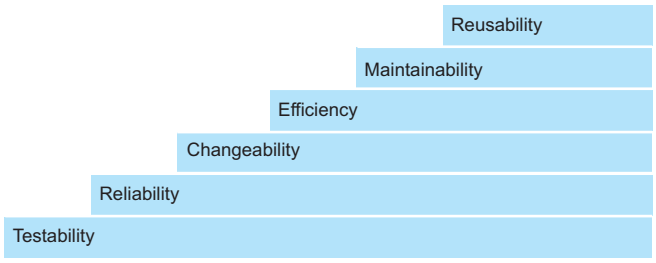
Since we have used a generic approach, it is possible to add other layers to satisfy other needs such as code portability and/or code security.

Two other complementary and more detailed levels are derived from the quality characteristics (or quality factors) of the first level.
A second level of sub-characteristics is then defined. A third level of control points on the software source code completes the model.

The standard Goal/Question/Metrics (GQM) from Basili[8] was applied to build the additional 2 layers. The selected control points in any assessment depends on the context, in particular the programming languages used as well as the execution environment of the software.

The existing literature on the subject provided an extensive set of candidate control points, in particular the very complete taxonomy established par Diomodis

Figure 2 : The SQALE Quality Model in its Standard Representation



Spinellis[10]. Figures 3 and 4 depict some elements of the complete quality model.

When necessary, sub-characteristics corresponding to sub-activities of the source life cycle have been defined. In this way, the Testability characteristic is broken down into the Unit Testability and the Integration Testability. In the same manner, Maintainability is broken down into Readability and Understandability: read and/or navigation and understanding activities are considered to be the principal activities of a maintenance person when working on a piece of source code developed by another.

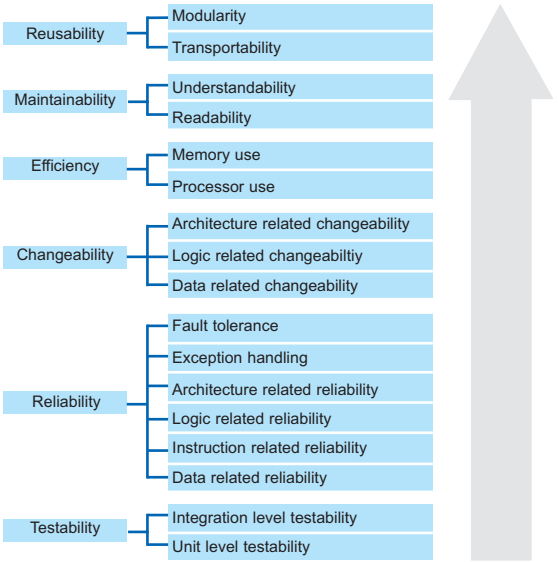
In this structure, a sub-characteristic or a control point used in the model is never duplicated in any higher level. It is used only once in the model. Using a control point and a sub-characteristic only once is an essential rule of the SQALE model, which must be followed at all times.

The consequence of this rule is that no testability criterion will appear in the Maintainability layer. The only sub-characteristics that remain attached to this characteristic are Readibility and Understandability.
The sub-characteristics such as Testability or Evolutivity traditionally associated to Maintainability in existing models are now previous characteristics of the SQALE model. It is a consequence of the fact that this characteristics are needed earlier in the lifecycle of a source code file.

A particular property of the SQALE quality model is that it is a requirements model. Each control point in the model is a quality requirement to be satisfied. This is in accordance with Ph Crosby[7]'s first "absolute concept": "Quality must be defined as conformance to requirements.
Consequently, the non conformance detected is the absence of quality".

Therefore the control points in the quality model are "Must have" that, when complied to, describe without controversy pieces of source code of high quality. The base requirements, the control points are

Figure 3 : Ranked Sub-Characteristic Example



either metrics with thresholds outside of which the source code is not compliant, or coding rules universally recognized as well founded and which do not allow any exceptions.

Also the requirements must be consistent. Indeed, it must be possible in practice to deliver a piece of software without any defect (non-compliance) when referring to the model's requirements.

The last property of the SQALE quality model is its capacity to support two different points of view, the developer point of view and the owner/user point of view, as described in figure 5.

By reading the model horizontally, non-compliances can be analyzed from the originating characteristic or sub-characteristic

Figure 4 : Some SQALE Quality Model Control points

Maintainability	Understandability	There is no function with an essentiel complexity ev(G) > 5
Maintainability	Understandability	There is no continuous instruction
Maintainability	Understandability	There is no break instruction outside a switch structure
Maintainability	Understandability	The comment percentage is > 25%
Maintainability	Readability	There is no file with more than a 1000 lines
Maintainability	Readability	There is no dead code

and one can understand the impact on the development activities (testing, implementing changes ...).

By reading the model vertically, the consequences for the owner of operating the software with non-compliances can easily be understood.

The SQALE Analysis Model

The following discussion illustrates the fact that the analysis model is the most "touchy" element of a product assessment method.

In a hypothetical project, let's suppose that a project leader wishes to rate the maintainability of his/her application software using an overall grade.

In order to compute the overall grade various basic grades must be aggregated.

This aggregate must be computed on two dimensions:

- In the dimension of the application's structure, the overall Maintainability grade of the application results necessarily from the Maintainability grade of each of its components and sub components (for example, each class and each method of each class).
- In the dimension of the ranked quality model, the sub-characteristics (Readability and Understandability) which contribute to the Maintainability characteristic are aggregated.

Currently, many assessment reports are built during each aggregate step by computing the grade as an average of the component grades.

In many cases, the grades are weighted by trying to take into account the relative priorities of sub-characteristics or the size of each assessed file.

This commonly used method, supported by many analysis tools, is not satisfactory since it produces erroneous diagnoses and consequently leads to wrong decisions.

In the hypothetical project, one of the requirements for Maintainability is that the source code is sufficiently commented. In the context, the threshold for an acceptable level of comments is fixed at 25% per file. If half of the software source code has a comment ratio of 15% and half of the code has a comment ratio of 40%, then an overall grade computed by averaging base grades will deliver a ratio of 32,5%, which therefore will be taken as an acceptable result for the overall maintainability of the application.

Sadly, in practice, the additional comment ratio in one part of the software will not help in any manner to maintain the part where comments are missing.
Each piece of software, if it is to be maintainable, must have enough comments. Such issues do not happen with the SQALE analysis model.

As stated previously, the quality model is a requirements model. The way it has been built ensures the quality targets a total absence of non compliances. As written by Ph. Crosby, assessing a software source code is therefore similar to measuring the distance which separates it from its quality target.

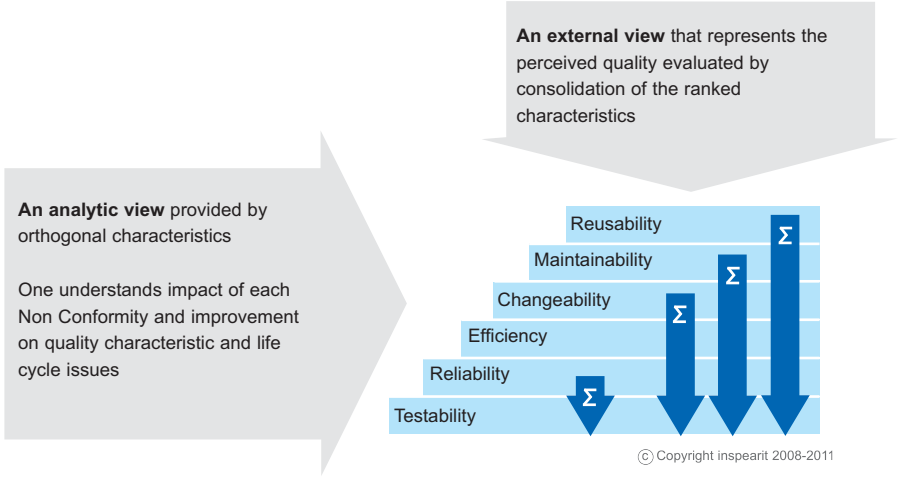
To measure this distance, the concept of remediation index has been defined and implemented in the SQALE analysis model.

An index is associated to each component of the software source code (for example, a file, a module or a class). The index represents the remediation effort which would be necessary to correct the non-compliances detected in the component, versus the model requirements.

Since the remediation index represents a work effort, the consolidation of the indices is a simple addition of uniform information, which is a critical advantage of our model.

A component index is computed by the simple addition of the indices of its ele-

Figure 5 : The two Viewpoints of the SQALE Quality Model



ments. A characteristic index is computed by the addition of the base indices of its sub-characteristics.

Many code reviews are based on aggregating averages. This approach is unsatisfactory since it produces erroneous diagnostics that may lead to wrong decision-making processes.

A sub-characteristic index is computed by the addition of the indices of its control points.

Base indices are computed by rules which comply to the following principles:

- A base index takes into account the unit remediation effort to correct the non-com pliance. In practice, this effort mostly depends on the type of the non- compliance.

  For example correcting a presentation defect (bad indentation, dead code) does not have the same unit effort cost as correcting an active code defect (which implies the creation and execution of new unit tests, possible new integration and regression tests).

- A base index also considers the number of non-compliances. For example, a file which has three methods which need to be broken down into smaller methods because of complexity will have an index three times as high as a file which has only one complex method, all other things being equal.

In the end, the remediation indices provide a means to measure and compare non compliances of very different origins and very different types. Coding rule violation non-compliances, threshold violations for a metric or the presence of an antipattern non-compliance can be compared using their relative impact on the index.

Practical Use

The SQALE quality and analysis models have been used to perform many assessments of software source code, of various sizes and in different application domains.

The same layered quality model has been used to assess Cobol, Java, embedded Ada or C++ software source code.
For Java, C++ and Ada, the quality model contains, of course, object-oriented metrics to assess Testability, Evolutivity and Reusability.
The quality model also provides control points to detect the absence of antipatterns such as those identified by Brown[10].

The indices are computed based on the average remediation efforts estimated by the development team.

The index thresholds providing a rating in five levels (from “poor” to “excellent”) are established by the application managers.
Several graphs and diagrams are used to efficiently visualize the strengths and weaknesses of the assessed source code:

Figure 6 : Overall Application Index Dashboard

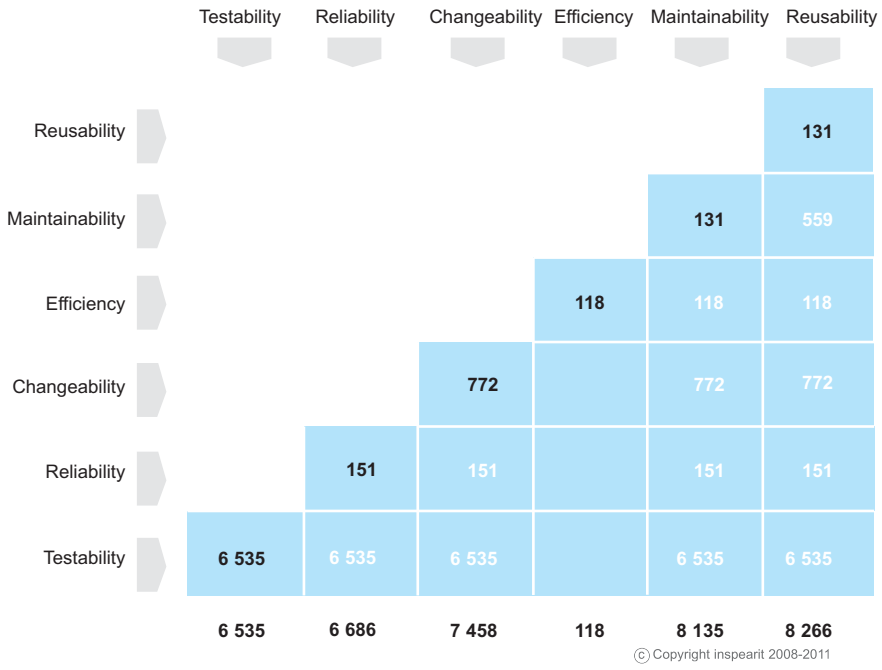
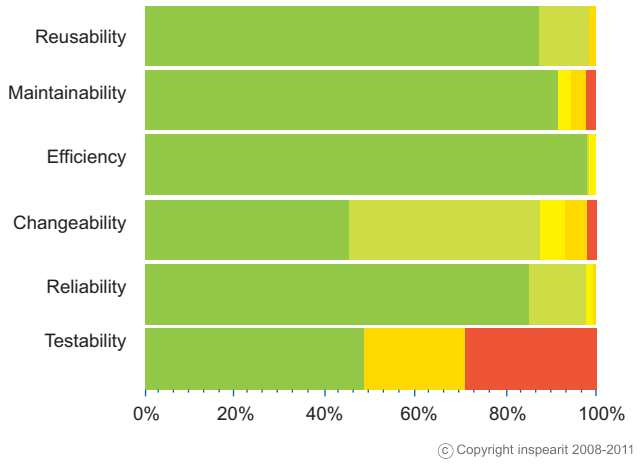


Figure 7 : File Distribution Diagram based on the Indices for each Characteristic



- Which software pieces contribute the most to the risks.
- Which quality criteria are the most impacted.
- What is the impact of the identified non-compliances on the project or its users.

Some of the indicators we use in our assessments and that provide in our experience a good visibility and good information for taking decisions are presented here. (Figures 6 and 7)
In the case of the application shown in this example, the Testability index is high. Therefore, the other external characteris-

tics cannot be satisfied (even partially), even in the presence of a rather good compliance to the control points for the higher characteristics.

The diagram presents a finding which may seem paradoxical. Indeed, to markedly improve the Maintainability of the application, as perceived by the owner, it is preferable and more efficient to improve the Testability than to add comments.

The analysis indicators mentioned have allowed us to localize precisely the high priority modifications (quick-wins) to apply to increase the Testability, and therefore, the overall quality of the application.

The analysis of the index density as shown in figure 8 for example (the density is the ratio of the index to the size of the piece of software) has also enabled us to compare the non compliance distribution between Reused, Modified and newly Created source code files. An additional effort is needed to improve the testability of modified files for example by re-factoring.

Conclusion

More than a quality model, we propose a standard approach for developing quality models derived from the software lifecycle. This approach enables us to precisely specify the expectations on software source code.

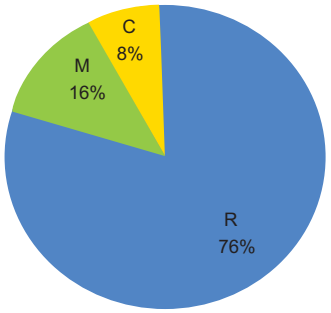
The resulting models remove several weaknesses in current reference quality models. The quality models implemented using the SQALE method present two similarities with the CMMI process model:

- On the one hand, like the CMMI, they organize the requirements in levels organized by priority.
- On the other hand, like the CMMI model which may be used to assess capability or maturity, the SQALE quality models may be used in an analytical manner or from the point of view of an owner or user of the source code.

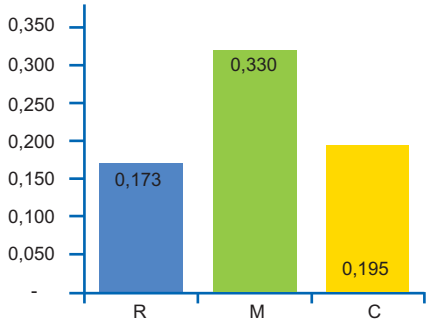
Such similarities of the models and the assessment methods allow us to perform joint product/process assessments in a fast and efficient manner with high added-value results to the development or owner teams.

Figure 8 : Index Distribution and Density Graph

TESTABILITY	R	M	C
Unit level Testability	4 130	970	495
Integration level testability	835	105	-
TOTAL	4 965	1 074	495



Index Distribution



Index Density by v(G)

Références

1. SEI, CMMI for Acquisition Version 1.2, CMU/SEI 2007-TR-017, Carnegie Mellon University, Pittsburg, 2007
2. SEI, CMMI for Development Version 1.2, CMU/SEI-2006-TR-008, Carnegie Mellon University, Pittsburgh, 2006
3. ISO/IEC, International Standard, «ISO/IEC 15939: 2007 (E), Systems and software engineering – Measurement process, 2007
4. Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., and Merrit, M., Characteristics of Software Quality, North Holland, 1978
5. McCall, J. A., Richards, P. K., and Walters, G.F., Factors in Software Quality, The National Technical Information Service, No. Vol 1, 2 and 3, 1977
6. ISO, International Organization for Standardization, «9126-1:2001, Software engineering – Product quality, Part 1: Quality model», 2001
7. Crosby, P. B., Quality is free: the art of making quality certain, New York: McGraw-Hill, 1979
8. Victor Basili, Software Modeling and Measurement: The Goal/Question/ Metric Paradigm. Technical report. University of Maryland at College Park, 1992
9. Diomidis Spinellis, Code Quality The Open Source Perspective, ISBN: 0-321-16607-8, Addison-Wesley, 2006
10. Brown et al, Anti Patterns: Refactoring Software, Architectures and Projects in Crisis, ISBN:978-0-471-19713, John Wiley, 1998

About inspearit

inspearit helps customers to improve the efficiency of their IT-related processes and systems. We assure safety and integrity of business critical operations and maximise return on investment to meet business goals.
The business drivers for delivery of effective software dependent systems have never been stronger. Rapidly, changing technology and complex legacy systems combined with outsourcing and regulatory compliance pressures raise the level of challenges and opportunities. By adopting a risk-based approach, inspearit helps companies to industrialise their software and systems processes (SPI).

Contact :

Alain Bloch
alain.bloch@inspearit.com

Le Visium
22 avenue Aristide Briand
CS 90001
94742 Arcueil Cedex
France
Tel : +33 (0)1 49 08 58 00