



QStudio® Concepts

QA Systems

The Software Health Company™

www.qa-systems.com

August 2003

Version 1.1.1

Introduction

QA Systems – The Software Health Company™ - is focused on improving its customers' *software health*. QA Systems develops software tools to assess, support, monitor and control the health (quality) of software applications developed by its customers both from both a preventative viewpoint (supporting good programming practices) as a diagnostic viewpoint (diagnosing possible software code health risks).

QStudio® products help companies to

- Reduce time to market by cutting testing time due to earlier detection of (potential) software errors
- Significantly reduce review effort (and therefore cost) by automating a major portion of the inspection process (adherence to coding standards, usage of best programming practices)
- Improve quality control and thereby code quality by directly supporting adherence to improved programming practices and corporate coding standards and avoid software quality degradation
- Assess the quality of its existing code base in order, for example, to establish maintenance budgets or for insourcing or outsourcing contracts

QA Systems' has developed QStudio® technology. QStudio® allows for the automated assessment, monitoring and control of higher-level software quality concepts. For the first time, concepts such as reliability, maintainability and efficiency are amenable to automated assessment. It allows software development quality control practices to be tied both deeper and earlier into the software development process. The benefits are better (management) insight into code quality, higher quality code with less human review effort, fewer errors and ultimately faster time-to-market.

QStudio® technology couples advanced parsing capabilities to the ISO 9126 quality standard framework. Furthermore, it makes use of a *pattern language* as a vehicle for knowledge capture and transfer (to the developer) on what best programming practices are for the particular language under consideration. Finally, its usage model is based on Total Quality Management Techniques (TQM) supporting a tight tie-in to quality assurance practices.

QStudio for Java Pro is the single user version of the QStudio product family. QStudio for Java Pro gives developers the ability to automatically assess and control software quality concepts such as reliability, maintainability and efficiency. Users have the capability to create new rules as well as customizing existing rules.

QStudio for Java Enterprise is an enterprise strength collaborative web-based solution that significantly improves software quality (and thereby productivity) by empowering all team members to automatically inspect and control the quality of Java based source code. QStudio for Java Enterprise allows software process managers (project managers and QA managers) and software developers to automate, coordinate and support quality control through a distributed software quality control and management system.

QStudio for Java Enterprise provides (in addition to all single user Pro functionality):

- Enterprise-wide control and reporting on quality metrics (ISO 9126 conforming) with respect to compliance of the source code to quality standards
- Code conformance to selectable project/team/departmental related code quality conformance standards
- Trend analysis based on formal milestone creation and reporting showing the quality evolution
- Code quality metrics overviews

QStudio for Java Enterprise automates formal enterprise quality control as opposed to the single user approach supported through QStudio for Java Pro.

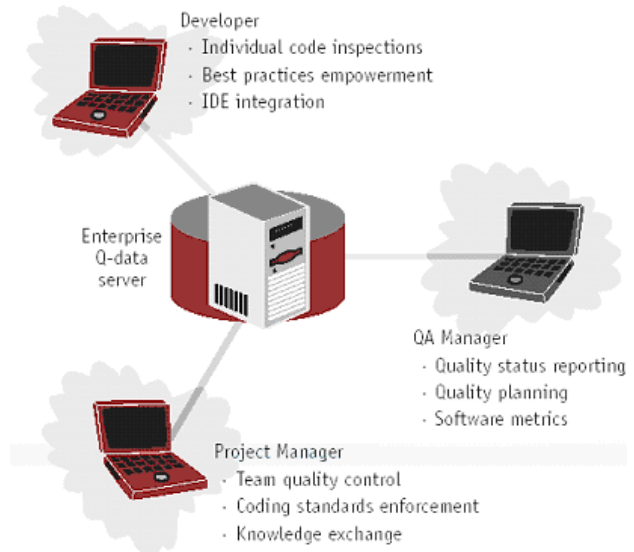
The Enterprise version is role based distinguishing software engineers, quality managers, project managers and application managers. Each role is granted different (related) functionality.

Multiple coding standards and quality standards configurations can be managed and made available to the software development team.

Formal inspections are supported. A formal inspection creates a milestone in the quality repository.

QStudio® for Java – The Software Health Tool for Java™ - products seamlessly integrate with all major Java Development Environments including JBuilder (from Borland), JDeveloper (from Oracle), Eclipse from Eclipse.org, WebSphere Studio (from IBM) and VisualAge for Java (from IBM). Sun ONE (from Sun) and NetBeans from NetBeans.org and IntelliJ are available later in 2003.

QStudio® for Java is available on the Windows 98/2000/NT/XP/ME, RedHat Linux 6.1 and higher, SuSE Linux 7.0 and higher and Solaris 6.1 and higher.



1 Concepts

No computer language is perfect. Every language has its pro's and con's. As the programming language community over the years gains programming experience, the application of language features and the typical language weaknesses and risks become visible through "best programming practices" which identify and specify language idiosyncrasies through programming tips and tricks and programming do's and don'ts. For major programming languages it is fair to say that best practices are usually fairly mature and documented.

No developer is perfect. Novice developers may not know the language well or lack extensive programming experience. Experienced developers may not be informed on the latest best practices or may have only recently entered that particular programming language domain. Even experienced programmers may (under severe deadline duress for example) break good programming practices.

QStudio® technology checks source code warning for error prone programming constructs. It makes use of world class programming practices to support the developer with knowledge and examples for code improvement. QStudio® is a software quality control tool. Both the novice as well as the expert developer will find it equally useful to assure and improve the quality of their code (and thereby the quality of the related software application) as well as to improve their language knowledge and programming skills.

QStudio® is based upon the following domains of expertise:

- *Pattern Languages* being the carriers of best programming practices and a vehicle for knowledge transfer
- *ISO/IEC 9126 Quality Model*
- *Total Quality Management* as a mature methodology for quality planning, quality control and quality improvement.
- *Static Analysis* using advanced parsing technology for source code quality analysis.

QStudio makes use of sophisticated parser generator technology that supports attribute grammar generation, symbol table generation, code walk algorithms and code generation generation.

QStudio® provides information at three levels of abstraction:

- *Rules* specifying the programming practices that should be applied as well as information on possible improvements. QStudio allows users to create new rules as well as to customize existing rules.
- *Observations* indicating the rules being violated and at what locations in the source code. Observations also specify the related *quality attribute* affected as well as the possible *quality risk impact level* on the code
- *Patterns* which provide the developer with a deeper understanding of the principles involved and the reasons why the rules being violated should be applied.

1.1 QStudio® and Pattern Languages

A best practice is a piece of applicable knowledge offering a solution to a certain problem or challenge in a certain context. A pattern is a method for documenting a best practice of an algorithmic nature.

Patterns were introduced by the building architect Christopher Alexander in the “The Timeless Way of Building” (1979). Within the software community the use of patterns became known with “Design patterns: Elements of reusable object-oriented software” by Gamma et al (1995). A software design pattern description typically takes one or two pages.

A software design pattern typically has at least the following elements or sections:

- **Name:** name of the pattern.
- **Problem Description:** describes when to apply the pattern and the problem and its context.
- **Forces:** describes the often-contradictory considerations to be considered when choosing a solution to the problem.
- **Solution:** describes the solution to the problem.
- **Consequences:** describes the results and trade-offs of applying the pattern.
- **Examples:** Sample Code or Improved Code sections

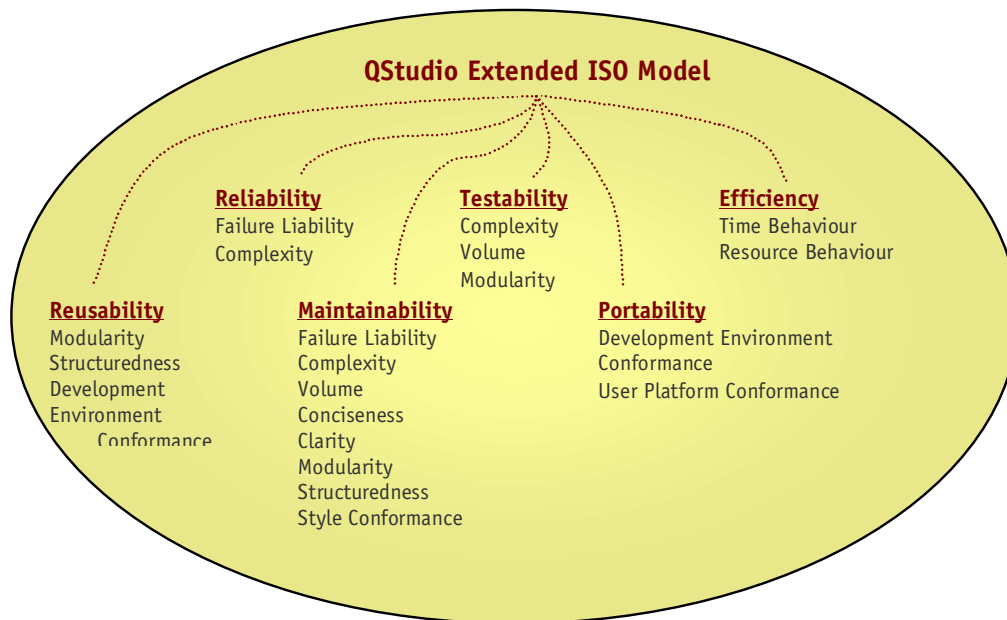
A *pattern language* is a structured collection of patterns that build on each other. QStudio® Uses such a pattern language.

QA Systems has collected and established an extensive database of patterns. The patterns act as input for the design of new rules. Sets of QStudio® rules are based on *Themes* that are topics of programming treated at a higher level of abstraction and specified using the pattern format. One example of such a theme is exception handling.

1.2 QStudio® and the ISO 9126 Quality Standard

QStudio® specifies quality concepts in a measurable way based on an extended version of the ISO 9126 quality standard. QStudio recognizes quality attributes such as reliability, maintainability, testability, re-usability, portability and efficiency.

The model defines a stepwise refinement of the notion of code quality into a set of ISO defined quality attributes and from these a further breakdown into quality sub-attributes. QA Systems proprietary technology maps these attributes in turn onto programming constructs.



Quality is measured through *code metrics based checks* as well as *rule based checks*. QStudio® rules incorporate quality attributes (in addition to more traditional code metrics such as cyclomatic complexity and static path count, for example) thereby providing a far more fundamental and advanced type of code quality metric than is currently addressed in competing quality control products.

QStudio® uses a three level quality attribute model based on ISO 9126. The model defines a stepwise refinement of the notion of *code quality* into a set of ISO defined *quality attributes* and from these a further breakdown into *quality sub-attributes* that in turn are measured through *metrics*.

QStudio proprietary technology defines the relevant mappings between attributes and sub-attributes as well as the mappings between sub-attributes and relevant metrics represented through QStudio Checks.

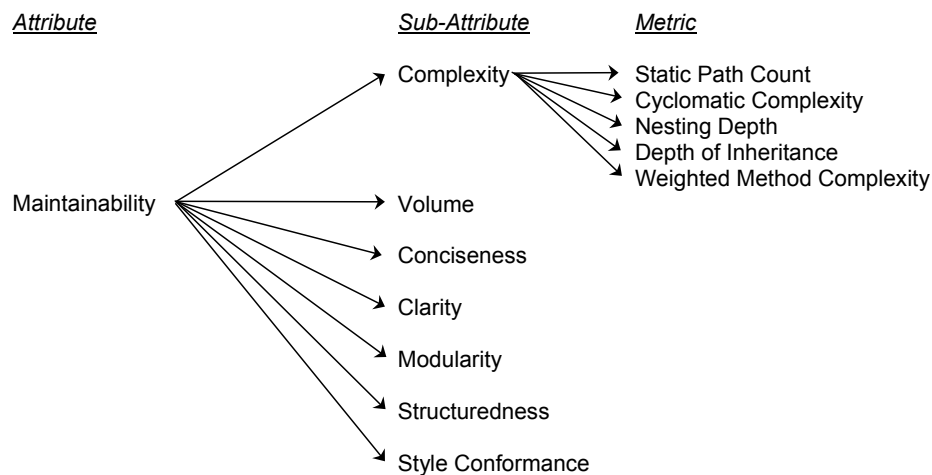
1.2.1 Quality Attributes

The following Quality Attributes are recognized:

- **Reliability:** The ability of a software product to keep operating over time without failures that renders the system unusable. Observations generated on reliability indicate a risk that the application will fail during actual use.
- **Maintainability:** The aptitude of the source code to undergo repair and evolution. Observations generated on maintainability indicate that changes are hard to implement.

- **Testability:** The amount of test resources needed to reach acceptable test coverage. Observations generated on testability indicate that a great number of test cases might be needed to reach acceptable test coverage.
- **Re-usability:** The suitability of the source code to be used by a variety of users. Re-use is the practice of using parts of source code that already have been developed. Observations generated on re-usability indicate that re-use is limited or difficult.
- **Portability:** The ability of the source code to be used on various user environments and development environments. Observations generated on Portability indicate risks that the software product can't be used on specific user platforms or that the source code can't be used "as is" in specific development environments.
- **Efficiency:** The ability of the software product to perform its functions related to the amount of resources that are used by the application. Observations generated on efficiency indicate that resources can be more effectively used.

In the figure below a part of the quality attribute model is presented elaborating the quality attribute *maintainability* into its various *sub-attributes*, the associated *metrics* are also shown for the *complexity* sub-attribute.



ISO 9126 Quality Attribute Model

1.2.2 Quality Sub-Attributes

The following quality sub-attributes are recognized:

Failure Liability: The possibility or probability that a product failure occurs as a result of the applied coding practice. Observations with this sub-attribute indicate that there is a risk that the software product will fail during use.

Complexity: Source code properties that offer great difficulty in understanding, solving, or explaining. Observations with this sub-attribute indicate that it may be hard for developers to understand the source code.

Volume: Source code is contained in methods, methods are contained in classes and classes build up the software product. QStudio counts the number of source lines, methods and classes. In a well-balanced software product, the number of source lines within a method and the number of methods within a class do not exceed certain limits. Observations with this sub-attribute indicate that source code quantities may be out of proportion.

Conciseness: The ability of source code to be marked by brevity of expression or statement free from all elaboration and superfluous detail. Observations with this sub-attribute indicate that there is a risk that the code is hard to understand because of superfluous statements.

Clarity: The ability of source code to be fully revealed or expressed without vagueness, implication, or ambiguity leaving no question as to meaning or intent. Observations with this attribute indicate that there is a risk that the intent of the code is hard to discover.

Modularity: The property of source code to be constructed with standardized units for flexibility and variety in use. Modularization of source code is established to facilitate re-use and to assign clearly defined interfaces to software parts. Observations with this sub-attribute indicate that the used code construct may reduce the modularity of the source code.

Structuredness: The property of source code to be constructed with an interrelation of parts in an organized whole. Observations with this sub-attribute indicate that the internal language structures may not be adhered to or that there may be a mismatch between the defined code construct and the underlying software structure.

Style Conformance: Adherence of the source code to a convention with respect to a particular manner, form, or technique by which the code is created. Observations with this sub-attribute indicate that there may be a more accepted and elegant way (best practice) of coding.

Development Environment Conformance: The ability of source code to be used in similar development environments. Observations with this sub-attribute indicate that there is a risk that the source code cannot be used in other development environments without rework.

User Platform Conformance: The ability of source code to be used in different user environments". Observations with this sub-attribute indicate that there is a risk that the software may not operate on some platforms.

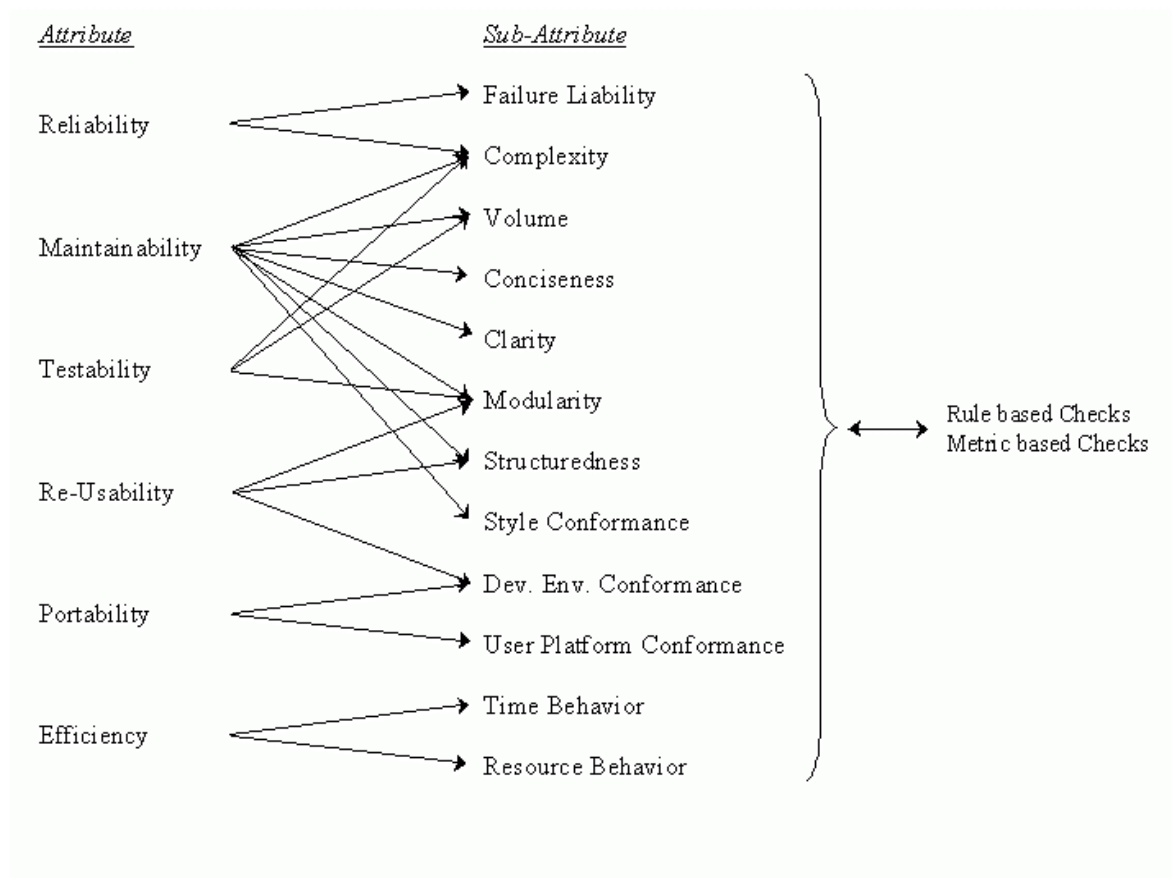
Time Behavior: The properties of the source code with respect to response, processing times and on throughput rates in performing its function. Observations with this sub-attribute indicate that the used code construct may be less time efficient.

Resource Behavior: The properties of the source code with respect to the amount of memory, processor time, file or network resources used in performing its function. Observations with this sub-attribute indicate that computer resources may be less efficiently used than possible.

1.2.3 Attribute Relationships

The QStudio quality attribute model supports the notion that a sub-attribute may be related to multiple attributes. This is a fact of life that bears upon the deeper concepts of quality. For example, the sub-attribute *complexity* relates to the attribute *Reliability* because complex code usually contains more defects than less complex code, in addition *complexity* relates to the attribute *Maintainability* because complex code is harder to maintain. Finally, the sub-attribute *complexity* relates to the attribute *Testability* because complex code requires more test cases to reach sufficient test coverage.

The figure below sketches this notion:



1.3 QStudio® and Static Analysis

Static analysis is an automated technique to review source code and recognize certain code constructs. QStudio makes use a sophisticated parser generator technology (a functional that supports attribute grammar generation, symbol table generation, code walk algorithms and code generation generation).

QStudio also supports PMD (pmd.sourceforge.net) specifications allowing:

- QStudio users to define their own rules via the PMD specification
- QStudio users to tap into the PMD user community and make use of the various rule sets defined by the PMD community
- PMD users to seamlessly extend their rule sets with the ISO quality model by importing them into QStudio and adding the ISO Model attributes

A great number of different programming constructs can be recognized and tested on compliance to good programming practices by means of a so-called QStudio® *Check*.

Examples of checks (in Java) are checking that a `catch` clause should contain at least one statement or checking that the static path count of a method is not too high (avoiding code complexity).

Each check provides a description of the good programming practice compliance violation (*rule*) as well as advice as to what the associated good programming practice could be (*observation*). The rule and the observation together indicate how the relevant programming practice is violated as well as how the code can be adapted (improved) to satisfy the relevant programming practice.

There are two types of checks: rule based checks (i.e. checks that result in observations on coding practices) and the metric based checks (i.e. checks that result in observations on code quality metrics values).

QStudio static analysis capability allows a seamless integration of code quality metrics and code improvement. During a static analysis run for Java, for example, the actual values of a set of international standardized Java code metrics (e.g. Static Path Count, Cyclomatic Complexity, Nesting Depth, Depth of Inheritance, Weighted Method Complexity, Comment Density, Coupling Between Objects and Lack of Cohesion) is evaluated for each of the methods/classes.

These values are used by the metric based checks to generate observations depending on the user defined rating values (maximum or minimum allowed values). An observation is generated each time a method/class has a metrics value outside the allowed range.

The quality of the code is assessed by counting the number of generated observations (hits). The code quality is lower when the number of hits increases. Rule-based checks are also used for code quality assessment. Each of the rules is related to a code quality characteristic. As an example the check: "Variable names **j**, **k**, **m**, **n** and **i** should be of type **int**" is clearly related to the *clarity* of the code as following naming conventions makes code easier to understand.

For the metric based checks, counting the number of hits assesses the quality of a specific quality characteristic. The quality attribute tree serves as a classification structure for both the rule based checks and the metric based checks.

Each check is assigned to a quality sub-attribute, for example, the check: "If a variable should never change it should be assigned **final**" is assigned to the sub-attribute *clarity*, and the

check: "The Static Path Count exceeds the maximum allowed level for this method" is assigned to the sub-attribute *complexity*.

Multiple checks are assigned to each of the quality sub-attributes. Employment of a three level quality attribute model during code analysis has a number of advantages: tying the checks to the quality sub-attributes show the developer, in a concise and unambiguous way, how an observation is related to the (detailed) quality characteristics of the code.

Improving the code by following the advice given, it is clear how the code quality improves. It is also clear that by counting the number of (sub-) attribute hits, a detailed quality characteristic can be assessed quantitatively for (a batch of) source files.

1.4 QStudio® and Total Quality Management

QStudio® makes use of the Total Quality Management (TQM) model developed by W.A. Shewhard, W.E. Deming and J.M. Juran. TQM is a well-accepted and mature approach towards quality assurance and is today applied across all major industries worldwide.

The following TQM concepts are used within QStudio®:

- Quality (definition)
- Multi customer focus
- Quality control
- Continuous quality improvement
- Quality and customer focus

TQM defines "Quality" in a broad sense and applies it to products (objects) as well as processes (activities). Quality is not just "Freedom of deficiencies" but also "Meeting customer needs" and in turn the customers are defined as "All that are affected by the quality characteristics of the product or service both internally (people who develop the product) as externally (people who use the product or service)".

TQM states that quality should be specified using objective quantitative indicators. As a result of multiple users having multiple needs, one single indicator cannot express the level of quality. For example, the manager of the software development department would like the software to be easily maintainable to keep down maintenance costs whilst the end user of the software product is interested in usability and performance.

Impact levels are established by distinguishing three stakeholders: the *end user* (the person using the application), the *software process owner* (the person responsible for the - quality of the - software development process) and the *software developer*. Impact levels indicate to what extent the requirements of each of the stakeholders are satisfied.

QStudio® defines five impact levels with level 5 being the most severe:

- **Level 1:** This level relates to effectiveness of the software developer. Level 1 observations point to situations where best practices show that there is a better

alternative. Modifying the code according to the advice given will make the code more elegant.

- **Level 2:** This level relates to the effectiveness of teamwork. Level 2 observations point to a risk that the code is difficult to understand by peer developers because the code is unclear, complex, sparsely documented etc. Modifying the code according to the advice given will make it easier to (re) use and maintain by others.
- **Level 3:** This level relates to the (cost) effectiveness of the software development process. Level 3 observations point to a risk that the development efforts take longer than expected due to unforeseen problems, mismatches, inconsistencies etc. Modifying the code according to the advice will lead to more predictable and efficient software development.
- **Level 4:** This level relates to the effectiveness of the user experience. Level 4 observations point to a risk that parts of functionality cannot be used or that basic product features like performance, resource behavior, user friendliness, accuracy, compliance are not within acceptable limits. Modifying the code according to the advice given will improve the quality of the end user's experience.
- **Level 5:** This relates to effectiveness of the application. Level 5 observations point to a risk that the application may fail. Modifying the code according to the advice given will improve the reliability of the application.

Quality Control is performed to assure the quality of a product (or process). It is based on a feedback loop and consists of the following steps:

1. Evaluate the quality of the product.
2. Compare the outcomes to quality goals.
3. Act on the difference.

QStudio® enables *automated quality control* on source code. The corporate code quality goals can be put in a coding standard using QStudio® rules. The coding standard specifies which rules need to be applied and what their parameterizations are. The developer uses the QStudio® to apply the coding standard to evaluate the source code and, in the event of identified non-compliances, performs rework (guided by the observations, rule descriptions and patterns that QStudio® provides)

Another key concept of TQM is *continuous quality improvement* represented more familiarly as a continuous iteration of Check, Learn and Improve activities (the CLI wheel).

The QStudio® CLI wheel involves three major activities that are successively performed by the developer:

- **Check:** QStudio® is used to analyze the source code and check if the code does not violate a selected rule set (*view*).

- **Learn:** the developer examines the analysis results, reads the associated rule descriptions and patterns and learns why the code would be improved and how this can be done.
- **Improve:** the developer modifies the code incorporating the suggestions and observations made by QStudio®.

Code improvement can take place along various axes or *views*:

- **Coding Standard:** a coding standard can be used as a baseline for improvement. The code will improve by making it comply with the defined coding standard.
- **Impact level:** the code will improve by reducing the hits at a certain quality risk impact level, starting with level 5 (most severe) and working down to level 1.
- **Quality Attribute:** the improvement can be focused towards certain quality attributes. For example, the number of maintainability hits can be reduced.
- **Theme:** the code can be improved by choosing a topic such as ‘Exception Handling’ and reducing the number of hits on this topic.