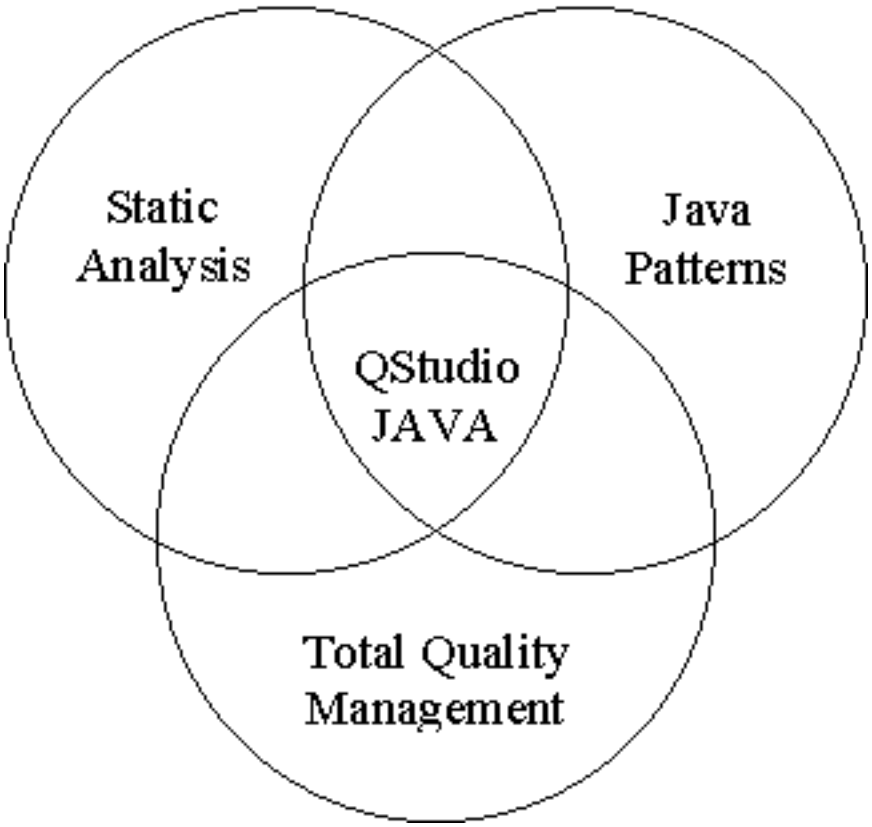
Concepts of QJ-Pro

1. Introduction

No programming language is perfect. Every language has it's pro's and con's. As the world over the years gains programming experience, the application of language features and the typical language weaknesses and risks become visible. Then the "best programming practices" arise, specifying the tips, tricks, do's and don'ts of the language. For Java over the past years numerous books and articles have been written and it is fair to say that the Java best practices are currently well documented.

No Java developer is perfect. Junior developers might not know the language very well or lack experience. Experienced developers might not be informed on the latest best practices or enter a new type of application domain. Even very skilled programmers might not be concentrated for a while and introduce code errors.

QJ-Pro is a static analysis tool that checks the developers source code, warns for error prone programming constructs and uses the world's Java best programming practices to support the developer with knowledge and examples for code improvement. QJ-Pro is both a safety net and a learning tool. The junior as well as the experienced developer can use it to improve the quality of their source code (and thereby the quality of the software application) and to improve their own Java knowledge and programming skills.



QJ-Pro usage

QJ-Pro is based upon three fields of study.

- Static Analysis using modern compiler technology for source code checking.
- Java Pattern Languages being the carrier of Java's best programming practices and a vehicle for knowledge transfer.
- Total Quality Management as a mature methodology for quality planning, quality control and quality improvement.

Each of the disciplines are well known and respected. They are integrated and build in QJ-Pro to provide a fully-fledged solution for code checking and code quality improvement.

2. Static Analysis

Static analysis is an automated technique to "walk through" the source code (available in *.java files) and recognize certain code constructs. A great number of different code constructs can be recognized and each of these are implemented by means of a QJ-Pro "Check". Checks are based on "Rules". A rule defines a good programming practice and the

related check can determine if the rule is applied. If the rule is not applied , QJ-Pro will issue an "Observation" message. Example:

- Check: 59
- Observation: "The body of a catch clause should contain at least one statement."
- Rule: "An exception usually signals undesired or unexpected results during program execution. If such a situation occurs, appropriate measures should be taken, and these measures have to be programmed in the catch clause. It is good practice to print at least a warning message. It helps finding problems in your program."

In addition to the rule based checks, there are checks based on code quality metrics. The most commonly used Java code metrics are evaluated and if (configurable) rating values are exceeded, an observation message is issued. Example:

- Check: 50
- Observation: "The static path count of a method, i.e., the upperbound of all possible paths through a method, should be less than 20."
- Rule: "The static path count is directly related to testability of software. The smaller it is, the easier it is to test your program."

In the example, the rating value is set to "20". This value can be altered by the user of QJ-Pro.

The checks are identified with a number ("59"). This number also identifies the observation and the rule because each unique check is based on a unique rule that will generate a unique observation (a one to one to one relationship). Further on in this document we will use this number to identify the check, the related observation and the related rule (e.g. check 59, observation 59 and rule 59).

QJ-Pro contains many checks and new checks are added each release. The developer can manually turn checks, or groups of checks, on or o ff. When the source code is analyzed by QJ-Pro (an analysis run) all selected checks are evaluated and if rule non-compliance's are recognized (a hit) an observation message is generated.

The observation and the rule indicate the programming practice and ' how the code can be improved. In addition there is a link to the quality characteristics of the source code and the quality of the resulting software application. These are expressed using "Impact Levels" and "Quality Attributes". An Impact Level and a Quality Attribute is assigned to each rule. An Impact Level (1 thru 5) expresses the need and the effect of improvement. The Quality Attribute indicates the quality characteristic that is related to the rule. Example:

- Rule 59
- Impact level = 4 (Software User); this means that the user of the software application might encounter strange and unexpected behavior.
- Quality Attribute = "Failure Liability"; this means that there is a chance that the rule

non-compliance results in a software error.

The Impact Levels and Quality Attributes are an effective tool to judge if a certain rule should be implemented and to set priorities if there are many observations. They provide a link between the individual code construct and the behavior and characteristics of the resulting software application and the code as a whole. QJ-Pro furthermore allows rule selection using Impact Levels and Quality Attributes. An Impact level or Quality Attribute can be switched off. All checks that relate to the Impact Level or Quality Attribute are then disabled.

3. Java patterns

Best practices can be characterized by (many) parts of practical knowledge, each offering a solution to a certain problem or challenge in a certain context. A Pattern is a popular and well-accepted method for documenting best practices. Patterns were introduced by the building architect Christopher Alexander in the book "The Timeless Way of Building"(1979). Within the computer society, the use of patterns became popular with the book "Design patterns: Elements of reusable object-oriented software" (1995) by Gamma et al. Over the past years many books and articles containing patterns have been written.

A pattern is a structured piece of text containing specific practical knowledge. Usually a pattern takes one or two pages. The pattern is structured by a fixed number of sections each containing a specific element. The following elements are essential for a pattern.

- Name; name of the pattern.
- Problem Description; this section describes when to apply the pattern and the problem and its context.
- Forces; this section describes the often contradictory considerations to be considered when choosing a solution to the problem.
- Solution; this section describes the solution to the problem.
- Consequences; this section describes the results and trade-offs of applying the pattern.

Some other sections can be added for specific types of patterns. Java patterns usually contain sections with code examples: "Sample Code" and "Improved Code". A special kind of patterns that deal with programming practices are sometimes called "Idioms". In the remainder of this document the term pattern includes the idiom type. A "Pattern Language" is a structured collection of patterns that build on each other. QJ-Pro contains a pattern language.

QJ-Pro makes use of patterns in two ways. In the first place, the use of patterns plays a key role in the development of QJ-Pro. The developers of QJ-Pro have a database of patterns at their disposal. All published (Java, C++, Smalltalk) patterns books, articles on patterns and patterns internet sites are brought together in a knowledge database. Newly published books

and articles are added regularly. The published patterns act as input for the design of new rules. Each set of new QJ-Pro rules is based on a "Java Theme". This is a topic of Java programming, treated on a higher level of abstraction and specified using the pattern format. This theme pattern is designed for QJ-Pro and is based on existing published patterns and represents the world's knowledge and experience on Java programming. Example:

- Theme Pattern: Exception Handling
- Related Rules 9, 56, 57, 58, 59 and 63

Secondly patterns are used to transfer programming knowledge to the user of QJ-Pro. The theme patterns are available in the QJ-Pro Rules Guide.

QJ-Pro provides information and knowledge on three levels of abstraction.

- The observation provides brief information on what rule is disregarded and at what location in the source code. The related quality attribute and impact level indicate what software quality characteristic is affected and what quality related risks are involved
- The rule specifies the programming practice that should be applied and information on improvement.
- The pattern provides the developer with a deeper understanding of the principles involved and the reason why the rule should be applied.

Knowing and understanding patterns will increase the quality of the code and improve the developer's skills. Kent Beck, author of the book "Smalltalk Best Practice Patterns", explains the use of patterns as follows.

"To my mind, most programming is not about going up on the mountaintop, being struck by lightning, and bringing down "the answer" carved on stone tablets. Most programming is applying yesterday's solutions to today's problems. Most of the time a client is convinced I am a genius, I've just applied a pattern that they don't know. I'm not a great programmer, I'm a pretty good programmer with great habits."

4. Total Quality Management

The features of Qstudio for Java are based on the principles of Total Quality Management (TQM) as developed by W.A. Shewhard, W.E. Deming and J.M. Juran. TQM is an accepted and mature approach towards quality management and applied in all branches of industry. There are many "guru's" that contributed to the development of TQM. Although their practical implementation might differ they share the same conceptual foundation. The following TQM concepts are used within QJ-Pro.

- Quality (definition)
- Multi customer focus
- Quality control

- Continuous quality improvement

5. 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 process both internally (people who develop the product) as externally (people who buy and use the product)". Furthermore TQM states that quality should be specified using objective quantitative indicators.

As a result of the multiple users having multiple needs, one single indicator can not express the level of quality. The manager of the software development department would like the software to be maintainable to cut down on maintenance costs while the actual user of the software product is more interested in useabilty and performance.

These concepts are adopted within Qstudio for Java. Quality is specified in an objective and measurable way using a "Quality Attribute Tree". Quality is measured using hit scores on metrics based checks and rule based checks. Note that by assigning quality attributes to the rules, in addition to the traditional code quality metrics (e.g. cyclomatic complexity, static path count etc.), a new type of metrics arise. As an example the number of hits on rules that are related to "Structuredness" is a measure for the structuredness of the code.

The impact levels are defined by distinguishing three customers: the Software Product User (the person using the (Java) software), the Software Process Owner (the person in charge of the Java software development process) and the Software Developer (the person writing Java code and using Qstudio for Java). The impact levels indicate how well their needs are satisfied.

6. Quality Control

This activity is performed to assure the quality of a product (or process). It is based on the feedback loop, and consists of the following steps.

- Evaluate the quality of the product.
- Compare the outcomes to quality goals.
- Act on the difference.

QJ-Pro enables quality control on source code. The code quality goals are put in a coding standard using the QJ-Pro rules. The coding standard specifies which rules should be selected and how they must be configured within QJ-Pro. The developer uses these settings to evaluate the source code and performs rework (guided by the observations, rule descriptions and patterns) if there are non-compliance's. Example of a coding standard.

Rules Included:

- Impact level: 5, 4, 3
- Quality Attributes: Reliability, Maintainability, Efficiency

Rules Excluded

- 16: "Any variable that is initialized and never assigned to should be declared final."
- 131: "Replace magic numbers by more readable symbolic names."
- 137: "Use spaces rather than tabs to indent your program text."
- 139: "Surround branches of an if statement with braces, even if it contains only a single statement."
- 147: "Provide javadoc comments for each public declaration."

Average maximum number of hits allowed per file:

- Impact level 5, 4: max: 0 (no hits allowed)
- Impact level 3: max: 0.5

Metrics Configuration:

- Configure check 50 "Static Path Count" to 100
- Configure check 51 "Average Static Path Count" to 20

Before software testing takes place, the Quality Engineer will use QJ-Pro to check if the code meets the coding standard.

7. Continuous quality improvement

One of the key concepts of TQM is continuous quality improvement. As early as 1939 WA Sheward defined the Plan-Do-Check-Act (PDCA) circle for continuous quality improvement. Later on this approach is more familiarly known as the "Deming wheel". The Deming wheel was designed for teams but can, as suggested by I. Masaaki, also be used for the individual developer.

This approach is taken in the QJ-Pro CLI Wheel. Continuous improvement involves three major activities that are successively performed by the developer.

- Check; Qstudio for Java is used to analyze the source code and check if the code does not violate a selected rule set.
- Learn; the developer examines the analysis results, reads the rule descriptions and patterns attached and learns why the code should be improved and how this can be done.
- Improve; the developer changes the code.

The tree activities define an improvement cycle that is performed continuously. Each cycle further improves the quality of the code.

QJ-Pro offers features to support the developer in performing the CLI Wheel activities.

Various approaches for code improvement are possible.

- Coding Standard; a coding standard can be used as a baseline for improvement. The code will improve by making it comply with a coding standard.
- Impact level; the code will improve by getting rid of hits on an impact level, starting with level 5 and working down to level 1.
- Quality Attribute; the improvement can be focused towards quality attributes. As an example the number of structuredness and clarity hits can be reduced.
- Java themes; the code can be improved by choosing a Java topic such as "Exception Handling" and reducing the number of hits on this topic.

An essential characteristic of the CLI Wheel is that each improvement step lasts. The quality of the Java code primarily depends on the expertise of the developer. A lasting improvement of the code quality can only be achieved by increasing the know-how of the developer. The learn activity therefor is an essential part of the CLI Wheel. The developer must have an in depth understanding of the coding rule and the reason of improvement. Otherwise the same inadequacy will reoccur.

8. Quality Impact Levels

QJ-Pro supports the software developer in making high quality Java code. Checks generate a list of observations pointing to parts or lines of Java code that can be improved. The nature and significance of these improvements vary over a wide range. There are observations that point to improvements on programming style and -habits while others indicate risks on program failure. The Quality Impact Level, ranging from 5 (most) to 1 (least), indicates how significantly the i quality of Java code can be improved by implementing the rule that is related to the observation. A Quality Impact level is assigned to each of the rules and presented with the observation. It is suggested to work down the list of improvements and start with the highest level.

Significance of improvement is related to the parties that are involved and share in the benefits of quality source code. Assuming the situation that software developers participate in software projects that develop software for users, the following three parties can be recognized.

The Software Developer spends many hours in writing source code. The code quality for this party is determined by the features of the programming language and in addition (s)he is interested in coding best practices that make programming effective, efficient and enjoyable.

The Software Process Owner will try to setup a software development process that is effective, efficient, flexible and predictable. The quality of the software process, among other factors, is influenced by the quality of the code . The code quality for this party is determined by the ability of fast and error free integration with other software parts and minimizing the

time needed for debugging and rework. In order to optimize teamwork and hand over code ownership, the code should be clear and easy to understand.

The Software Product User will use the product as a personal instrument. The quality of the software product is largely determined by the quality of the code . The code quality for this party is determined by the ability to offer the required features and functionality and to be reliable and fail safe.

The parties are not equally important to satisfy. The Software Product User, bringing in the money, is most important. The Software Process Owner, offering reasonably priced software within a limited development period, is of secondary importance and the Software Developer, doing a professional job stands on the third place.

The above specified code quality requirements boil down into 5 increasingly important levels. Impact Levels have been assigned "worst case". If an observation covers more Impact Levels, e.g. a code construct might cause integration problems (level 3) , is hard to understand (level 2) and there is a smarter way of doing it (level 1), then the highest level is assigned. The advice given on improvement is downwards consistent. In the aforementioned example, the advice will provide in a smarter code construct that rules out integration problems and is easy to understand.

9. Definitions of impact levels

The Quality Impact Levels are defined according to the 5 levels.

Impact Level 5: "Software Failure" This is the most significant level because it relates to the basic use of the software product. Level 5 observations point to a risk that the product fails operation and can't be used. The program may stop or show no response. The code is largely improved by changing the code according to the advice given.

Impact Level 4: "Software User" This is an important level because the user assumed quality of the software is not satisfied. Level 4 observations point to a risk that parts of functionality can't be used or that basic product features like performance, resource behavior, user friendliness, accuracy, compliance are not within acceptable limits. Changing the code according to the advice given will increase the user's judgment of the product quality.

Impact Level 3: "Software Process" 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. Improving the code will lead to predictable and efficient software development.

Impact Level 2: "Development Team" This level is related 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, hardly documented etc. Improving the code makes it easier to (re)use and maintain by others.

Impact Level 1: "Developer" This level is tight to the professionalism of the software developer. Level 1 observations point to situations where best practices show that there is a better alternative. Better in a sense that it is more elegant, easier to apply, smarter, shorter etc. Improving the code will increase it's beauty and make the programmer more effective and efficient as a result of the learning curve.

Impact Levels have been assigned "worse case". If an observation covers more Impact Levels, e.g. a code construct might cause integration problems (level 3) , is hard to understand (level 2) and there is a smarter way of doing it (level 1), then the highest level is assigned. The advice given on improvement is downwards consistent. In the aforementioned example, the advice will provide in a smarter code construct that rules out integration problems and is easy to understand.

10. Quality Attribute Tree

QJ-Pro provides facilities for quality assessment and quality improvement of Java code . International quality standards and guidelines, and common quality management techniques are used as a basis for the quality related features of QJ-Pro.

The term "quality" has a very broad and general meaning and can't be used without refinement. ISO/IEC 8402 defines it as " the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs". By nature,"quality" is determined by a number of "characteristics" that each contribute, in some extend, to the over all level of quality. The most difficult tasks are: 1) to capture the characteristics of source code in a well defined and consisted quality framework and 2) to make code quality measurable.

The most commonly used technique to set up a quality framework is the composition of a so called "Quality Attribute Tree". The quality characteristics of an entity (e.g. cars, machines, services, software etc.) are defined in a coherent set of leveled attributes and metrics. QJ-Pro uses a three level quality attribute tree. The tree defines a stepwise refinement from the general overall characteristic "Code Quality" into a more detailed set of quality attributes, e.g., Reliability, Maintainability, Testability..... The quality attributes are further refined into subattributes e.g. for Maintainability: Complexity, Conciseness, Modularity, Structuredness..... and finally quality metrics are attached to the subattributes e.g. for Complexity: Cyclomatic Complexity, Nesting Depth, Static Path Count..... These quality metrics have been extensively described in literature.

In the following example a part of the quality attribute tree is presented. It shows how the quality attribute "Maintainability" is elaborated into subattributes and metrics (only the

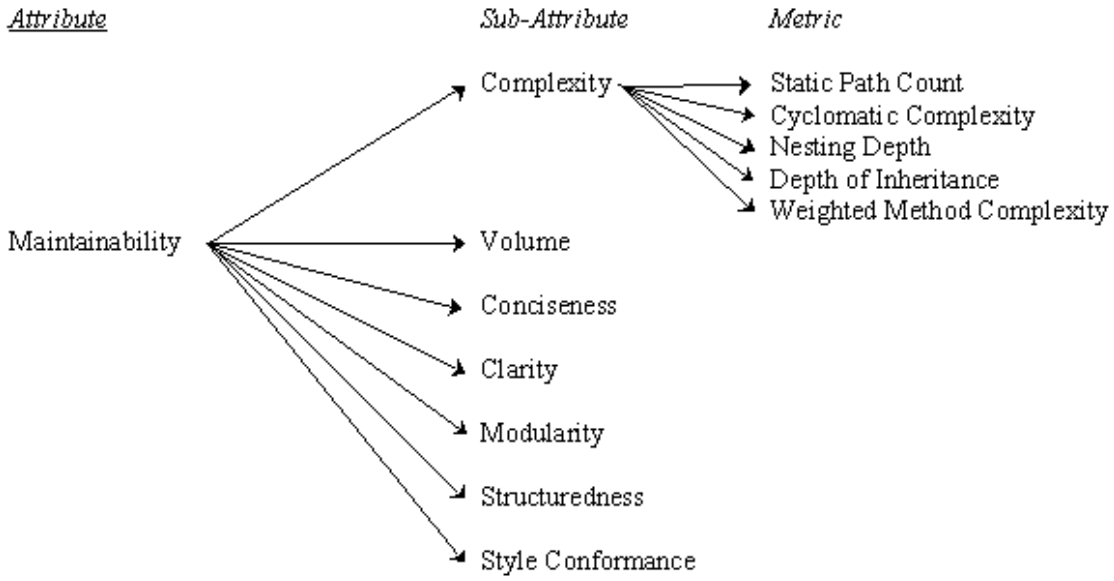
complexity metrics are shown).

Quality Tree

The advantage of such an approach is that the quality on multiple levels of detail can be assessed by measurement. Measuring the score on Static Path Count reveals detailed information on a specific type of Complexity. The developer might consider to change the code when the Static Path Count exceeds 100. Having the scores of all Complexity metrics makes it possible to assess the over all Complexity of the code. Results might show that Static Path Count, compared to Nesting Depth, contributes very little to the over all Complexity and restructuring initiatives might be started to improve the Nesting Depth. Finally the over all Maintainability can be assessed by analyzing the scores of all metrics that are related to the Maintainability subattributes. Again the balance between the various subattributes can be analyzed and improvement activities could be considered.

QJ-Pro's key feature of static source code checking has been used to provide a quantitative

assessment of the source code quality. The following picture shows the quality attributes and quality subattributes of the QJ-Pro quality attribute tree. The International Standard ISO/IEC 9126 has been taken as a baseline.



Quality Attribute Tree

The QJ-Pro quality attribute tree shows that, in addition to the above mentioned multiple subattributes to attributes relation, it is also possible (and needed) that a subattribute is related to multiple attributes. This is a fact of life that bears upon the deeper concepts of quality. As an example, the subattribute Complexity relates to "Reliability" because complex code usually contains more defects than less complex code, in addition Complexity relates to "Maintainability" because complex code is hard to maintain and finally Complexity relates to "Testability" because complex code requires more test cases to reach a sufficient test coverage. The "many to many" relationship between attributes and subattributes introduces no difficulties in using them for quality assessment or quality improvement. The quality is assessed top down by evaluating the attributes first and then the subattributes and metrics.

During a static analysis run, 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, 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 gets worse when the number of hits increases.

The QJ-Pro static analysis capability allows an integration of code quality metrics and code improvement. 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), where each observation is accompanied with an advice on improvement.

The rule based checks are also used for code quality assessment. Each of the rules is related to a code quality characteristic. As an example check 37: Variable names "i", "j", "k", "m" and "n" should be of type "int", is clearly related to the "Clarity" of the code. Name conventions make code easier to understand. As for the metric based checks, the quality of a specific quality characteristic is assessed by counting the number of hits.

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 subattributes. E.g. check 16: "If a variable should never change it should be declared "final" is assigned to the subattribute "Clarity", and check 50: "The Static Path Count exceeds the maximum allowed level for this method" is assigned to the subattribute "Complexity". Multiple checks are assigned to each of the quality subattributes.

Employment of a three level quality attribute tree for static code analysis has a number of advantages. Tying the checks to the quality subattributes shows the developer, in a concise and unambiguous way, how an observation is related to the (detailed) quality characteristics of the code. With improving the code by following the advice given, it is clear how the code quality improves. By counting the number of subattribute hits, a detailed quality characteristic can be assessed quantitatively individually and for a batch of source files. The quality attributes serve as an effective overall indicator. They are the most commonly used standard software quality characteristics (ISO/IEC 9126). The overall quality can be quantitatively assessed by counting attribute hits. QJ-Pro facilitates the reporting of subattribute hit totals and attribute hit totals.

11. Quality Attribute definitions

Reliability: -definition- "The ability of a software product to keep operating over time without failures that renders the system unusable". This is certainly the most feared attribute and relates to the stigma "bad quality" because we expect a software product to operate always. Any value less than 100% is hard to accept. Observations generated on reliability indicate a risk that the product will fail during actual use.

Maintainability: -definition- "The aptitude of the source code to undergo repair and

evolution". Once the source code is released, the maintenance stage is entered. Modifications to the source code are made as a result of growth, change of functionality or detected errors. This stage of the life cycle tends to be very resource consuming. This attribute relates to the amount of effort and time needed for maintenance activities. Observations generated on maintainability indicate that changes are hard to implement.

Testability: -definition- "The amount of test resources needed to reach an acceptable test coverage". The testing stages (which include problem solving) before release of software should provide some degree of certainty that the product has the required features and that it will operate without failures. An acceptable test coverage indicates that sufficient test cases are performed to be able to judge if the product can be released. This attribute relates to the amount of effort and time needed for (white box and black box) testing activities. Observations generated on testability indicate that a great number of test cases might be needed to reach an acceptable test coverage.

Portability: -definition - "The ability of the source code to be used on various user environments and development environments". Software products are usually required to be used on a variety of platforms. Although platform independence is Java's key feature, some code constructs could decrease it's ability to operate on different platforms. Portability is also important in cases of re-use. It should be possible to use the source code in various Java 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: -definition - "The ability of the software product to perform its functions related to the amount of resources that are used". A software product needs resources to perform it's functionality. Resources could be: time, memory, CPU capacity, files, I/O channels etc. This attribute relates to the actual amount of resources that are needed. Software products that need large amounts of resources have poor Efficiency. Observations generated on Efficiency indicate a that resources can be more effectively used.

12. Quality Subattributes definitions

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

Complexity: -definition- "Source Code properties that offer great difficulty in understanding, solving, or explaining". Observations with this subattribute indicate that it will be hard for developers to understand the source code. As a result it will be difficult, time consuming and error prone to add or change parts of the code.

Volume: -definition- "Source code quantities". Java source lines are contained in methods, methods are contained in classes and classes build up the software product. QJ-Pro 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 subattribute indicate that source code quantities are out of proportion. As a result it is difficult to gain a clear view of the source code and changes are hard to implement and error prone.

Conciseness: -definition- "The ability of source code to be marked by brevity of expression or statement : free from all elaboration and superfluous detail". To give the reader a clear picture of the meaning of of the source code it is important for programmers to be brief and to the point. Observations with this subattribute indicate that there is a risk that the code is hard to understand because of extra unnecessary statements. As a result it is difficult to gain a clear view of the source code and changes are hard to implement and error prone.

Clarity: -definition- "The ability of source code to be fully revealed or expressed without vagueness, implication, or ambiguity : leaving no question as to meaning or intent". The software developer provides parts of source code based on the (technical) design of the software product. This creative process is based on the knowledge and experience of the developer and the line of reasoning during programming. A peer developer that wants to understand the code has no other means than the statements itself to reveal the meaning. Observations with this attribute indicate that there is a risk that the intend of the code is hard to discover. As a result it will be difficult, time consuming and error prone to add or change parts of the code.

Modularity: -definition- "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 reuse and to assign clearly defined interfaces to software parts. Observations with this subattribute indicate that the used code construct reduces the modularity of the source code. As a result the code is harder to reuse and there is a risk that the interaction between software parts is unintelligible.

Structuredness: -definition- "The property of source code to be constructed with an interrelation of parts in an organized whole".There are two aspects to this subattribute. The Java language is an internally structured language and the developer has facilities to structure the code. Observations with this subattribute indicate that the internal Java structures are not adhered to or that the used code construct declines the software structure. As a result it is more difficult to understand the function of software parts or the relation between the various parts of a software product. In addition it is hard to extend the software product with new code to add functionality.

Style Conformance:-definition- "Adherence of the source code to a convention with respect

to a particular manner, form, or technique by which the code is created". The world expertise of Java programming increases over time. Experienced Java developers find new approaches and the programming mastery is refined. Observations with this subattribute indicate that there is a more accepted and elegant way of coding. As a result the source code conforms to the worlds best practices of Java.

Development Environment Conformance: -definition- "The ability of source code to be used in similar different development environments". In reuse situations, the developer uses parts of code that have been developed by others. It is possible that the used code is developed in another type of development environment. Observations with this subattribute indicate that there is a risk that the source code can not be used in other development environments without rework. As a result it will take some developers more time and effort to use the code.

User Platform Conformance: -definition- "The ability of source code to be used in different user environments". The software product is usually required to operate on a wide variety of platforms. Observations with this subattribute indicate that there is a risk that the software product does not operate on some platforms. As a result some users can not use the software product.

Time Behavior: -definition- "The properties of the source code with respect to response, processing times and on throughput rates in performing its function". The user expects the software product to react prompt and without needless waiting periods. Observations with this subattribute indicate that the used code construct is less time efficient. As a result processing- and response times are not optimal short.

Resource Behavior: -definition- "The properties of the source code with respect to the amount of memory, processor time, file or network resources used in performing its function".The software product needs computer resources to perform its functions. Observations with this subattribute indicate that computer resources are less efficiently used. As a result more computer resources are consumed than actually needed.

13. Definitions of terms

Check: a check is an action performed by the QJ-Pro Kernel. The check is either based on a rule or a metric. The check is performed on source code. The output of a check is an observation.

Metric: a way of assigning numbers to code features which is used to measure the Code Quality.

Rule: a mutually agreed standard approach towards programming.

Pattern: a piece of practical knowledge put in a structured format. Patterns offer solutions to

practical problems. Patterns are used to communicate programming knowledge to the developer, specially in cases where the developer offends general coding practices. The pattern is issued based on (a combination of) checks.

Observation: the visual result of the check. This can be a warning message (checks based on rules) or a value (checks based on metrics).