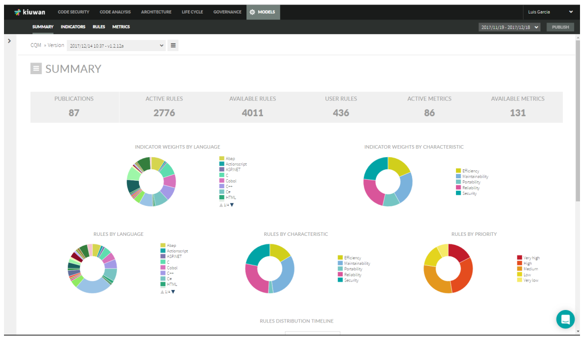
Models and CQM

This page introduces and explains Kiuwan Models and the concept of CQM (Checking Quality Model).

Contents:

- Introduction to CQM
    - What is CQM?
    - Why CQM?
    - ISO-25000 based
    - CQM benefits
- The CQM methodology principles
    - Structured model layer
    - Source code indicators layer
    - Technology indicators layer
    - Software characteristics indicators layer
- Static Analysis in Automated Software Quality Tests
    - Reliability
    - Efficiency
    - Maintainability
    - Portability
    - Global indicator layer: CQM indicator
- Understanding indicators
    - Comparison and evolution

Kiuwan offers the ability to manage models associated with the analyses carried out on the applications via the management utility, **Models Manager**.


kiuwan
CODE SECURITY
CODE ANALYSIS
ARCHITECTURE
LIFE CYCLE
GOVERNANCE
MODELS
Luis Garcia
SUMMARY
INDICATORS
RULES
METRICS
2017/11/19 - 2017/12/18
PUBLISH
CQM » Version
2017/12/14 10:37 - v1.2.12a
SUMMARY
INDICATOR WEIGHTS BY LANGUAGE
INDICATOR WEIGHTS BY CHARACTERISTIC
RULES BY LANGUAGE
RULES BY CHARACTERISTIC
RULES BY PRIORITY
Abap, Actionscript, ASP.NET, C, Cobol, C++, C#, HTML
Efficiency, Maintainability, Portability, Reliability, Security
Very high, High, Medium, Low, Very low
RULES DISTRIBUTION TIMELINE

PUBLICATIONS	ACTIVE RULES	AVAILABLE RULES	USER RULES	ACTIVE METRICS	AVAILABLE METRICS
87	2776	4011	436	86	131

To analyze an application, it is necessary to have and configure a **Model**. This is not an easy task. It requires minimum knowledge of the repository of hundreds of rules that help you validate the code and how to select and parameterize them. Configuring details for other Kiuwan indicators is also needed to fine-tune your analysis. Although these are complex tasks, Kiuwan helps you all the way with useful tools to narrow your needs.

Introduction to CQM

What is CQM?

CQM is a model for extracting analytics of a software product, designed by Kiuwan and available 'out-of-the-box' so that users can begin to analyze their code immediately. Once the methodology behind code certification is well known, users are able to "calibrate" their own models, creating them from scratch or from other existing models.

Before using it, let's introduce the concepts and methodology behind CQM, then, go into the details of the elements that compose a model in Kiuwan: Indicators, Rules, and Metrics.

Why CQM?

Kiuwan has a deep experience advising its customers to get better in software development. Kiuwan has noticed that its customers have common troubles and interests, so there was the need of defining a methodology that was supported by tools, products, and solutions. This methodology must guarantee that the solution adopted will provide the target benefits and will resolve the most common troubles. Having a defined methodology saves time and money in the software development process of these companies. When you cut time in implementation, you get extra benefits because you minimize the risks that surround this kind of project: long projects are risky ones.

ISO-25000 based

CQM is ISO-25000 based. It defines internal quality scope and characteristics. This standard provides a set of concepts in order to build a common language about quality.
ISO-25000 defines three techniques of validation for software products:

- **Internal Quality** is the set of characteristics of the software product from an internal view. Internal quality is measured and evaluated against the internal quality requirements. Details of software product quality can be improved during code implementation, reviewing and testing.
- **External Quality** is the set of characteristics of the software product from an external view. It is the quality when the software is executed, which is typically measured and evaluated while testing in a simulated environment with simulated data using external metrics.
- **Quality in Use** is the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment.

CQM benefits

CQM provides software analytics that allow to:

- Abstract from the technical layer. Your information will be independent of program languages and platforms.
- Compare different versions of the same software. This answers the most important question: has my software improved?
- Compare different applications. It does not matter if they are different kinds of applications or they are developed in different technologies.
- Evaluate the technical requirements in order to accept the software from a third-party provider.

And other benefits:

- Aggregation of data. You can aggregate the information from different applications in order to get an evaluation of the software produced by a provider, a country or IT area compared to others.
- Continuous improvement process. You can apply a control methodology in your software life cycle process.

The CQM methodology principles

Structured model layer

The CQM methodology extracts evidence from the software's code and configuration. The process continues upwards in order to obtain technology-independent indicators to build evolution, comparators, and aggregations.

Source code indicators layer

This layer is responsible for reading the source code and extracting technical evidence. It has to classify this evidence in order to analyze it. In this step, you have to prioritize the evidence found, to distinguish what is critical, major, low or informative. It defines the required level.

Technology indicators layer

It can be based on different categories according to the technology. (For example, you are not going to get usability defects in PL/SQL source code, because this language does not concern itself with the user layer nor the presentation layer.)

So you have to define technology categories for the evidence found in the previous layer and aggregate it using the priorities based on the severity of the problem. These categories must be defined by thinking about the software characteristics of the next layer. For example, the group of evidence related to threads or parallelization affects software efficiency.
The output of this layer is the technical indicator that allow you to compare software based on the same technology.

Software characteristics indicators layer

This layer standardizes the indicators generated in the previous layer to get one normalized indicator for each software characteristics defined by **ISO 25000 standard**. CQM implements ISO 25000 focusing on internal quality. In order to provide indicators that correlate with the software characteristic, CQM proposes the following indicators:

- **Security**. The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.
- **Reliability**. The capability of the software product to maintain a specified level of performance.

- **Efficiency**. The capability of the software product to provide appropriate performance relative to the number of resources used under stated conditions.
- **Maintainability**. The capability of the software product to be modified. Modifications may include corrections, improvements, or adaptability of the software to changes in the environment and in requirements and functional specifications.
- **Portability**. The capability of the software product to be transferred from one environment to another.

At this point you have technology-independent software indicators that allow you to analyze your application; even if you do not have specific technical knowledge.

Static Analysis in Automated Software Quality Tests

Software quality management solutions function with automated tests that use static analysis proces ses to generate software quality metrics.

With the ability to parse code in almost every commonly used programming language, static analysis is useful in assessing a key set of five software quality indicators: software system security, code reliability, code memory usage and efficiency, maintainability, and portability.

These indicators represent the most significant aspects of high-quality code, and it’s important for developers and investors to monitor the metrics that static analyses generate to ensure they are producing secure, reliable, maintainable, portable, and scalable code.

Security

Static analysis looks for security vulnerabilities in source code by referencing the most common vulnerabilities as defined by OWASP.

The parser is able to monitor code for vulnerabilities to data injection by looking at any queries or command statements that take input from users. It checks to see that every instance of user input is properly prepared to avoid data injection. If the code does not properly separate user input from data structures, the parser will report a vulnerability.

A static analysis engine looks at authentication vulnerabilities by checking that the code protects user authentication credentials by hashing or encrypting the stored authentication information. It looks for weak account management functions, such as password recovery processes, for simple password overwriting vulnerabilities. It verifies that the source code hides URL authentication information via URL rewriting algorithms and looks at session timeout procedures that often produce vulnerabilities to session fixation attacks.

When scanning for security vulnerabilities, the static analysis algorithm tags structures that contain sensitive data and verifies that all direct references to these objects require authentication for data to be passed to the caller. Similarly, the parser looks at indirect references to sensitive data objects and the data mapping process to ensure that the call requires user authentication. When the parser locates an instance of potential vulnerability, it will tag and report the instance and include it in a security metrics report. After scanning through all of the most common types of software vulnerabilities, it will generate a report that references all potential vulnerabilities it has identified.

Reliability

Static analysis is especially apt at looking at the reliability of all modules and methods of software.

The source of reliability in code is the ability to produce a predictable outcome given wildly varying inputs. In static analysis, the parser identifies specific methods and isolates them, emulating their behavior when ported into other software modules. Once isolated, the algorithm will generate arguments and read method outputs, expecting the outputs to contain specific information and formatting. It will raise flags for any anomalies in a method.

The algorithm will then aggregate anomalies and generate a report that points to specific methods that compromise reliability.

Efficiency

The efficiency of code is largely dictated by memory usage and data flow.

When performing static analysis, the memory usage of a program is not explicitly readable. By definition, static analysis is performed on code without actually instantiating an interpreter; thus, the parser needs to infer memory usage by looking at the code and data structures.

A common source of code inefficiency calls for entire data sets to be stored in active memory. This is especially common in legacy code and in systems initially developed for small data sets. The static analys is algorithm looks for any such explicit calls to store data sets in memory and reports them. It will also look for cases of lingering data stored in active memory – any call for data to be stored in active memory will prompt a search for a call to drop that data from memory after working and updating a database.

The report will suggest an iterative alternative approach to carrying out a memory-intensive task, but it’s important for a knowledgeable developer to look at such reports. Some explicit calls to stored data sets in memory are necessary and entirely manageable if, for example, the data set is known to be small (such as metadata).

Static analysis procedures will also identify duplicate code and multiple calls to commit the same data to memory simultaneously.

In most cases, the static analysis of source code will reveal and report important sources of software inefficiencies. It’s especially illuminating when working with extensive legacy code and data structures that weren’t initially designed to be scalable.

Maintainability

The maintainability of code is closely related to the complexity of code. If the source code is overly complex, it can be difficult to make changes and incorporate new functions and modules quickly.

Thus, the static analysis protocol looks for sources of code complexity to generate metrics that quantify code maintainability.

A source of program complexity that is easy for a parser to identify is duplicate code in sibling classes. If certain classes share a parent and share explicitly defined methods, the code is overly complicated – the two classes could inherit their shared method from a parent class. A parser can easily identify such cases by looking for duplicate methods.

Another source of complexity, method side-effects, are easy for static analyzers to look at. By parsing the arguments, output, instances, and the method code, the algorithm can find methods that create complications by implicitly affecting data flow.

By quantifying program complexity, a static analysis report helps developers look at the maintainability of their code and improve it by pointing to specific sources of complexity.

Portability

A static analysis report gives insight into the portability of source code by isolating and testing methods. Deprived of their native environments, the parser can predict how well methods would perform if ported.

Much as with reliability and maintainability, the portability of code lies in its complexity. The complexity metrics that a static analysis generates help developers quantify the portability of their code.

It’s important to understand the function of static analysis in any automated software quality test. It’s a wonderful tool, but developers should understand its strengths and limitations.

Global indicator layer: CQM indicator

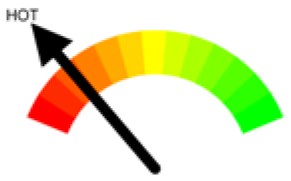
At the end of the quality process, you will get ONE standard and normalized indicator per software unit. Using this indicator you can extrapolate to evaluate groups of software applications; for example, evaluations of software providers, its areas and developers teams, etc.

Understanding indicators

An indicator can be defined as something that helps us to understand where we are, where we are going and how far we are from the goal. Therefore, it can be a sign, a number or a graphic and so on. It must be a clue or a pointer to something that is changing. Indicators are presentations of measurements. They are bits of information that summarize the characteristics of systems or highlight what is happening in a system.

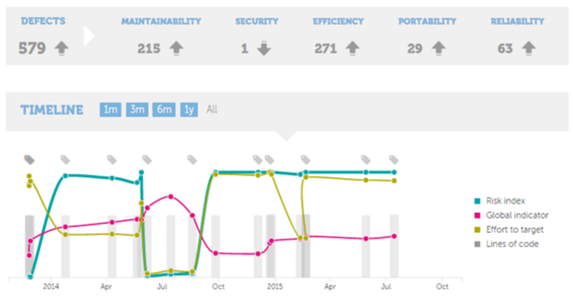
CQM indicators are normalized to represent these regions:

- 0-30 region. The characteristic pointed to by the indicator is in the RED zone. Improvements are needed.
- 30-70 region. Represented by YELLOW and means that you have to keep your mind on this indicator. Your next moves will depend on your requirements.
- 70-100 region. The GREEN zone. This is the zone where all indicators must be. No critical defects found.



Comparison and evolution

This normalization allows the comparison of different characteristics between them; this means that you can say if the software is more maintainable than it is efficient or reliable. You are going to compare different versions of the same application over time because the meaning of the indicator does not change.



DEFECTS	MAINTAINABILITY	SECURITY	EFFICIENCY	PORTABILITY	RELIABILITY
579 ↑	215 ↑	1 ↓	271 ↑	29 ↑	63 ↑