Continuous GQM: An automated measurement framework for the GQM paradigm

Diplomarbeit

von

Christoph Lofi

08 2005

AG Software Engineering
Fachbereich Informatik
Universität Kaiserslautern

Betreuer: Prof. Dr. H. Dieter Rombach, Prof. Dr. Philip Johnson,
Jens Heidrich, Marcus Ciolkowski

Erklärung

Hiermit erkläre ich, Christoph Lofi, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kaiserslautern, den 12. 08 2005

Contents

Abstract

Measurement is an important aspect of Software Engineering as it is the foundation of predictable and controllable software project execution. Measurement is essential for assessing actual project progress, establishing baselines and validating the effects of improvement or controlling actions.

The work performed in this thesis is based on Hackystat [1], a fully automated measurement framework for software engineering processes and products. Hackystat is designed to *unobtrusively* measure a wide range of metrics relevant to software development and collect them in a centralized data repository.

Unfortunately, it is not easy to interpret, analyze and visualize the vast data collected by Hackystat in such way that it can effectively be used for software project control.

A potential solution to that problem is to integrate Hackystat with the GQM (Goal / Question / Metric) [2, 3] Paradigm, a popular approach for goal-oriented, systematic definition of measurement programs for software-engineering processes and products.

This integration should allow the goal-oriented use of the metric data collected by Hackystat and increase it's usefulness for project control.

During the course of this work, this extension to Hackystat which is later called *hackyCGQM* is implemented. As a result, hackyCGQM enables Hackystat to be used as a Software Project Control Center (SPCC) by providing purposeful high-level representations of the measurement data.

Another interesting side-effect of the combination of Hackystat and hackyCGQM is that this system is able to perform fully automated measurement and analysis cycles. This leads to the development of *cGQM*, a specialized method for fully automated, GQM based measurement programs.

As a summary, hackyCGQM seeks to implement a completely automated GQM-based measurement framework. This high degree of automation is made possible by limiting the implemented measurement programs to metrics which can be measured automatically, thus sacrificing the ability to use arbitrary metrics.

Chapter 1

Introduction

This chapter introduces the background domain of this thesis: Measurement in Software Engineering. After that, the specific problems and goals being topic of this work are illustrated.

1.1 Measurement in Software Engineering

This section illustrates some aspects of measurement in the contexts of Software Engineering, it's importance, implications and problems.

A cite which became quite famous in the last years says *"you can neither predict nor control what you cannot measure"* [4]. Measurement is one of the steps needed to help the still young field of software development evolve from a state characterized by cryptic art and intransparent chaos to a state identified controllable, predictable, engineering-like processes.

The ability of a company to control and predict it's processes leads to major goals of all manufacturing and developing companies: Quality and efficiency. Measurement is an important aspect of gaining that control [4].

Measurement and metrics Measurement is defined as the "association of numbers with physical quantities and natural phenomena by comparing an unknown quantity with a known quantity of the same kind" [5]. The quantification of phenomena through the process of measurement relies on the existence of an explicit or implicit metric, which is the standard to which the measure is referenced [6]. A metric can be understood as a function m which assigns an property of a property set X to a value from the metric set M: $m : X \to M$. By doing this, different observed properties become *comparable*.

In software engineering, metrics usually have three different primary characteristics which can be classified as follows: The object they measure (*product* or *process*), the objectivity (*objective* or *subjective*) and the directness (*direct* or *indirect*).

In more detail [7, 8]:

Object (product / process)

Measurement is applied to a process or a product, resulting in either process or product metrics.

- A *product metric* applies to the characteristics of a intermediate or final *product* of software development. Examples of such a metric could be size, complexity or number of test cases.
- *Process metrics* measure characteristics of the development process itself. Examples could be number of found defects, overall effort or number of performed reviews.

Objectivity (objective / subjective)

- *Objective metrics* are based on absolutes measures and characterize a process or product in an objective way, so if repeated by a different individual, the result is the same. Most of the time, these metric are of numeric nature. Examples could be lines of code or number of found defects.
- The measurement of *subjective metrics* involve an human's subjective judgment. Naturally, two humans judging the same object independently don't have to end up with the same result. Examples are difficulty of a problem or estimated time needed to solve it.

Directness (direct/indirect)

- A *direct metric* is based on measurement of the product or process characteristic of interest. It does not rely on additional measurements or calculations. Examples are lines of code or number of issues.

- In contrast to that, *indirect metrics* involve the measurement of multiple characteristics. The final result is calculated based on these measured characteristics. Examples are productivity, effort or fault density.

As a final example, *source lines of code* is a *direct, objective product* metric.

Measurement During Software Development In *production* environments like manufacturing, the importance of measurement in process controll is well understood and many efficient and widely spread methods are in use. Nearly no production or manufacturing company takes the risk of not being able to control it's own processes [9].

In contrast to that, software companies are different. Although most of them are aware of the importance of better control and the need for measurement, many of them fail to implement controllable and measured processes.

Studies [10] found out that chaotic projects with no defined measurement and control during the development process fail more often or miss other project objectives. In numbers, this means that 3 in 10 projects are being canceled, 5 in 10 overrun their schedule and/or budget by nearly double, and only 1.6 in 10 finish in conformance with their deadlines and budgets. Ad hoc and uncontrolled processes represent a serious thread to a projects success.

Realizing this, Capability Maturity Model (CMM)[1] was developed to assess a companies ability to control their own processes. This assessment distinguishes five levels of maturity. These levels range from level 1 representing immature companies with no defined processes, measurement and control, followed by level 2 with simple, repeatable but still uncontrolled and unmeasured processes and ending after level 3 and at level 5 representing companies performing completely controlled, measured and optimized processes.

A report [11] from the Software Engineering Institute (SEI) shows that out of 542 software organizations participating in a CMM maturity assessment , 67%

[1]Capability Maturity Model, a method for evaluating on a scale from 1 to 5 how mature and controlled an organization is.

of them are at the lowest maturity level (level 1), and 20% are at maturity level 2.
This means that nearly 87% of all software companies expose themselves to the danger of not measureing and controlling their software development process described like before.

In contrast to that, the minority of software companies which was able to archive higher CMM levels and control and measure their processes tend not to fail their projects [12].

So, when controlling processes (and a crucial aspect of controlling, measuring) is so important and has such severe impacts, why do only a few companies implement functioning processes and measurement programs? The answer of that question is the topic of the next section.

1.1.1 Problems with Measurement and Control

As indicated by several researchers [13, 1, 14], the main reasons why organizations decide not to implement measurement programs are

- the lack of the needed competence
- lack of methods and tools needed for implementing useful measurement and
- effort linked with an actual implementation (definition of the program and measurement plans) and execution (measurement and analysis)

.

Most methods and tools known for controlling and measuring originate from a different field then Software Engineering. They are principles developed in manufacturing, production or other engineering fields. Unfortunately, it turned out that tailoring these known principles for software development is not as effective as hoped [14].

In most other engineering disciplines, the task of creating a new product is split into three main phases:

1. developing the product

2. developing the production process
3. execute and control the production process

Most measurement and control paradigms are tailored to support the third phase of this model. But when examining software as a product in this model, it turns out that all effort needed to create it is concentrated in the first phase (developing). After a software is developed, it's production is trivial (software can just be copied). So, in contrast to most traditional engineering products, software is not *produced, but* developed. To account for this aspect, specially tailored methods and paradigms have to be developed which consider the special situation of software development.

Up to today, a variety of software process improvement and/or assessment programs heavily based on measurement, such as Personal Software Process (PSP) [15], CMM [16], International Standard for Quality management systems (ISO9001) [17], Software Process Improvement and Capability dEtermination (SPICE) [18], Quality Improvement Paradigm (QIP) [14] or Goal/ Question/ Metric Paradigm (GQM) [2] have been created. But because Software Engineering is still a young field, many of these methods are still not widely spread, supported, tuned and accepted. And many of them are just methods - tool support which helps to implement them is usually limited.

This leads to the second problem: The implementation of measurement and process control programs is in most of the cases expensive and requires a high degree of discipline by all participants (which in most cases also includes project members on low operational levels like programmers), and as most data collection is manual, is prone to errors [19].

To many people, collecting and analyzing measurement data is a very tedious task which often seems not to pay off. Collecting data and analyzing data is often considered as superficial overhead by participants of measurement programs, mainly because often beneficial effects of measurement are not directly obvious. Especially in times of high pressures, managers tend to re-assign project members responsible for measurement to development activities [13].

One example for that is PSP [15] which in it's original form uses manual metrics collection. Every time a compilation error occurs, the developer has to stop his/her current work, and log on paper forms the details about the error. It is not only tedious, but also susceptible to bias, error, omission, and delay. This often leads to the abandonment of the program [19].

Though CMM does not prescribe how metrics should be collected and analyzed, it requires that all key process areas implement measurements to determine the status of activities. A study by Herbsleb, et al. [20] says that it took on average two years per level for a software development organization to get from CMM level 1 to level 3, and that the cost ranged from $500 to $2000 per employee per year. Quantitative measurement is explicitly addressed in CMM level 4, and Humphrey himself admitted that the greatest potential problem with the managed process (i.e. level 4) is the cost of gathering data, and that there are an enormous number of potentially valuable measures of the software process, but such data is expensive to gather and to maintain [16]. Due to high cost associated with metrics collection and analysis, it is a daunting task to apply measurement best practices to improve an software organizations development process in practice.

The summary this chapter is that if tools and methods for decreasing the difficulty and costs of actually implementing and executing measurement programs are available, this reduces the barrier preventing the establishment of measurement.

1.1.2 Measurement and Quality

Measurement is a core part of most quality improvement processes like QIP or SPICE. This section briefly introduces it's role in in these processes which will reappear frequently in the later chapters. .

The main purposes of measurement in context of improvement of software engineering related artifacts are [21]

- Assessment of the actual state of the artifact focused on it's strength and weaknesses
- Derivation of a "quantitative baseline", which means the characterization of an artifact not in a qualitative but in a quantitative way
- Determination of the effect of improvements by comparing the quantitative baseline with measurement results after the improvements

Summarized, the role of measurement in most improvement programs looks like this (illustrated in Figure 1.1):

For improving a given product or process, it is important to know it's state before the improvement program starts. This includes knowledge about which of the aspects are important and what their values are. Measurement is used here to define the *baselines*.

After the baselines are defined, certain controlling actions are taken. Their effect has to be assessed by a measurements which are compared to the baseline to provide feedback.

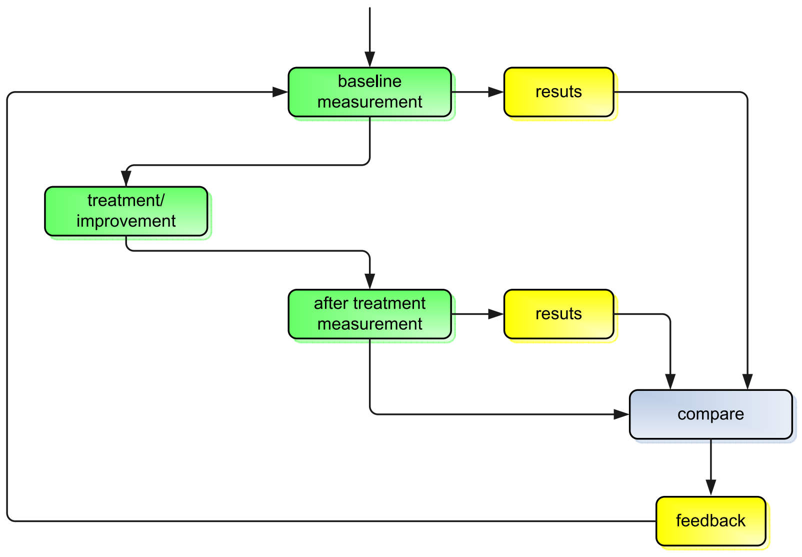


Figure 1.1: Measurement in quality improvement
Baselines are measured. After a treatment or controlling actions, a repeated measurement is conducted. The results can be compared to provide feedback.

1.2 Goal

A goal of this thesis is to

- develop a concept for adding GQM support to Hackystat
- create a reference implementation

- perform first evaluations of that reference implementation

The work done in this thesis is based on Hackystat [1], a fully automated measurement framework for software engineering processes and products. Hackystat is designed to *unobtrusively* measure a wide range of metrics. The actual measurements are performed by sensors which are connected to the software tools used during development (like code editors, build tools or configuration management systems). These sensors are able to perform *automated* and *unobtrusive* measurement. This architecture overcomes two of the common problems often associated with manually performed measurement: The high costs of actual measurement and the vulnerability to errors.

Hackystat's original goal was to serve as a academic platform for exploring and validating new software product and process metrics. But with it's powerful automated measurement abilities, it could also serve as a Software Project Control Center (SPCC), used for observing and controlling software development. Unfortunately, Hackystat's current design provides only limited support for data analysis suitable for project control.

The solution proposed in this thesis is to integrate GQM, a goal-oriented approach for defining useful and purposeful measurement programs.

By combining Hackystat with GQM, a goal-oriented, completely automated measurement system is created which limits itself to metrics which can be measured automatically by Hackystat. The concept of automated measurement and analysis based on GQM is also generalized under the term "cGQM" (continous GQM) in the later course of this work.

1.3 Structure

This thesis has four main parts: Introduction and concepts, cGQM and hackyCGQM and evaluations conclusions.

- Introduction and concepts:

  These chapters provide relevant context information needed to understand the content of this work. Single chapters can be omitted by readers already experienced with the introduced material. The following list of chapters can be used as a guideline for selecting an area of interest.

  - General introduction and measurement in software engineering environments (chapter 1)
  - Related works introducing GQM (2.1), QIP (2.2), SPCC (2.3) and Hackystat (2.4).

- cGQM and hackyCGQM: In chapter 3, cGQM is introduced, followed by it's reference implementation hackyCGQM in chapter 4.
- Chapter 5 provides different evaluations and their results.
- Chapter 6 concludes this thesis and indicated future possibilities and works.

Chapter 2

Related Work and Concepts

This chapter shows an overview of existing works and technologies on which this thesis is based upon or which are closely related.

These are, as illustrated in Figure 2.1:

- GQM (see section 2.1): A paradigm for goal oriented measurement for software products and processes. It is the foundation of continuous Goal/ Question/ Metric Paradigm (cGQM) (see chapter 3) which is the main topic of this thesis.
- Hackystat (see section 2.4): A framework for automated, unobtrusive measurement of software metrics. hackyCGQM (see chapter 4), the reference implementation of cGQM is integrated into Hackystat.
- QIP (see section 2.2): A paradigm for software quality improvement by cooperative and project learning. GQM and respectively cGQM can be used in this paradigm in a profitable way.
- SPCC (see section 2.3): A concept for controlling the software development process using measured data.

2.1 GQM

This section describes the Goal/ Question/ Metric Paradigm (GQM), a goal-oriented approach for setting up measurement plans for software engineering

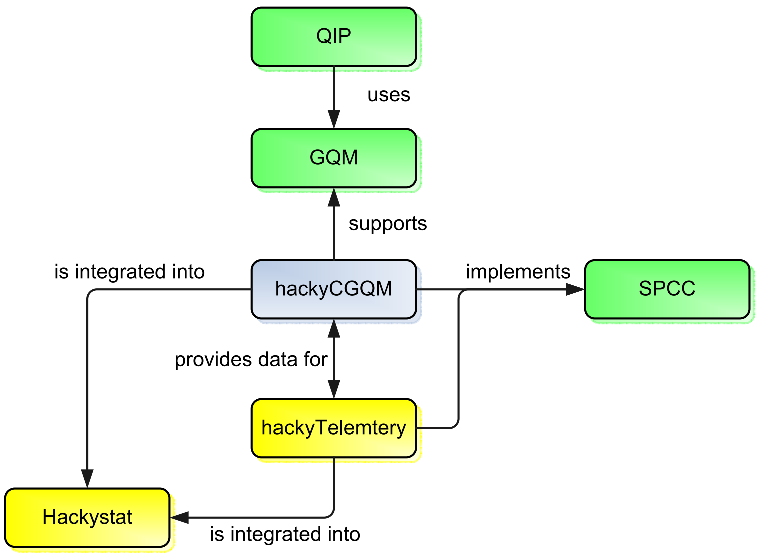


Figure 2.1: Relation between the relevant concepts and technologies
Yellow boxes indicate technologies *while green ones represent* concepts and methods.

related products and processes.

The goal of this section is to provide general insights into the basics of the GQM Paradigm. In later subsections, descriptions of the paradigm itself and basic techniques associated with GQM are provided.

GQM is of particular importance for this thesis as cGQM, this thesis's main topic, is based upon GQM.

The development of GQM started in 1984 at the University of Maryland [2, 3], additional research is done at the Technical University Kaiserslautern [21, 22] since 1992 and at the Fraunhofer Institute for Experimental Software Engineering Kaiserslautern (IESE) since 1996. In addition to that, several other groups are doing research on GQM.

The main goal of GQM is to develop rationale, traceable and efficient measurement strategies according to the underlying improvement or strategy goals.

The basic idea of GQM is that measurement should be goal-oriented, thus an early step while performing GQM measurement is to define a set of goals. These goals can be refined to questions which will help to solve the goals. Based on the goals and questions, metrics are precisely chosen. This procedure can be seen as a three layered structure, as illustrated in Figure 2.2.

In order to ensure efficient and rationale measurement, the GQM Paradigm provides a top-down approach for the development of measurement plans and a bottom-up approach for the interpretation of the recorded data.

By using this approach, there is an apparent justification why measured metrics have been chosen.

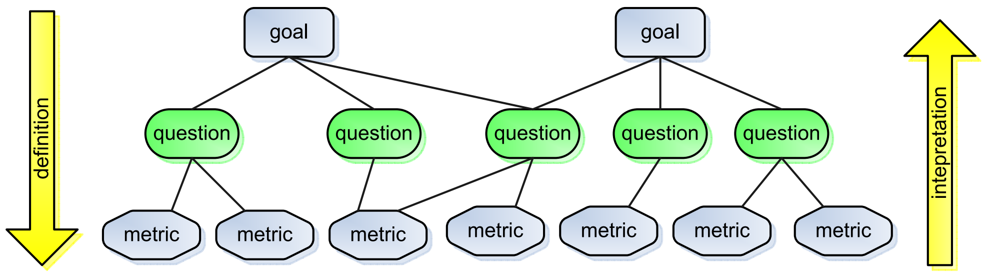


Figure 2.2: The GQM-Layers

Goals are refined to questions, these are refined to appropriate metrics

The next subsections will illustrate this concept in finer detail.

2.1.1 GQM Goals

One of the key concepts of GQM is that measurement is understood as a goal-oriented process. Defining exact and precise goals is fundamental. Therefore, this subsection introduces the definition of the term 'goal' in the context of GQM.

Usually, a GQM goal can be understood as a 5-tupel describing the five aspects (or dimensions) of a GQM goal: Object, Purpose, Quality Focus, Viewpoint and Context [22], as illustrated in Figure 2.3.

These dimensions are interpreted in the following way:

- The *object* describes the primary target of the measurement. Usually, this is the process or product which will be analyzed. Everything which can be measured or analyzed is a potential goal object, this includes documents like requirement documents or sourcecode, processes or parts of processes

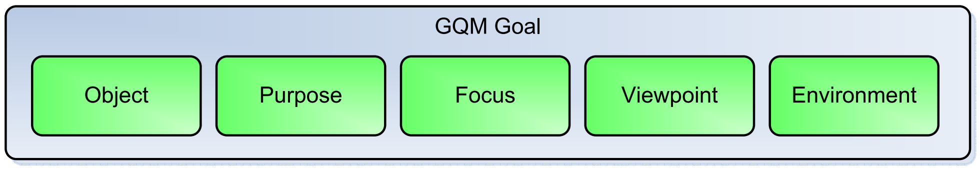


Figure 2.3: A GQM goal

Goals consist of 5-dimension-tuple

like the integration phase of the development process or products like the final, free-for shipping product.

- The *purpose* of the study describes the need to analyze the object. Different purposes could be, for example characterization, monitoring, evaluation, prediction or control.
- The *quality focus* describes which specific attribute or characteristics of the object should be the matter of concern. Possible focuses are for example, reliability, usability, correctness, costs or maintainability.
- The need and interpretation of data and information depends on the role and position of the people working with it. These different positions and role are grouped under the term '*viewpoint*'. Different viewpoints are for example programmer, project leader, tester, user or architect.
- The *context* describes the scope of the measurement program. This usually describes the programm's environment and setup and is so usable for knowledge transfer and determination of generalization.

An example for an possible GQM goal could be:

G1:

Analyze the *final product*
for purpose of *evaluation*
with respect to *usability*
from the viewpoint of an *inexperienced end user*
in the context of *Project X*.

After setting up the goals, they are usually characterized and examinated in more detail. A commonly used tool for that is the so called abstraction sheet, described in chapter 2.1.5.

The next step in the GQM process after defining the goals is setting up questions. GQM Questions will be the topic of the next subsection.

2.1.2 GQM Questions

GQM questions are created to refine and characterize goals. This subsection will give a brief overview of GQM questions.

Good questions should be designed in way that their answer will characterize a special aspect of their assigned goal. The collected answers of all questions assigned to a goal should provide a clear and usable overview of the goals.

Usually, a set of questions contribute to one goal, questions can be shared among goals.

Examples of questions which could be defined for the Example Goal 1 (2.1.1) are:

- Q1: How much time is needed to install the product?
- Q2: How much time is needed to fulfill a given test task?
- Q3: What is the subjective opinion of the test subject?
- Q4: What is the percentage of implemented functions covered by help texts and hints?
- Q5: To which degree does the product fulfill commonly accepted design guidelines (like standardized user interface guidelines or common design guidelines like the Nielsen Heuristics [23])?

Answering these questions give an incomplete insight into the goal (evaluation of usability). For answering these questions, measurement is needed. For doing proper measurement, according metrics have to be chosen.

Metrics are the subject of the next subsection.

2.1.3 GQM Metrics

Metrics describe a distinctive class of data representing a given property of the the object to be measured. Because GQM is a *goal-driven* approach, only those metrics are chosen for a measurement program that actually help fulfilling the stated goal. This stands in contrast to *metric-driven* approaches in which the focus is on measuring everything possible without a dedicated explicit goal.

2.1.4 Phases of GQM

Setting up goals, questions and metrics is only one phase in the complete process of implementing GQM. Altogether, four phases, as illustrated in Figure 2.4, are distinguishable in GQM [8]. Please note that this model does not prescribe in which order the phases have to be performed, although in many cases a strictly consecutive order is chosen.

Phase 1: In the first phase, the *Planning phase*, a project for applying measurement is selected, defined, characterized and planned. This results into a project plan.

Phase 2: Using the project plan, goals, questions and metrics are defined in the *Definition phase*, where goals are used to develop questions and questions to develop metrics.

Phase 3: After defining all needed metrics and using the information in the project plan, the actual data collection can be started. This phase is called the *Data collection phase* and results in a set of collected data.

Phase 4: The Analysis of collected data results into measurements, matching their according metrics. These measurements can be processed further, resulting in the answers of the questions which can be used to determine the fulfillment degree of the previous set up goals. This phase is called the *Analysis phase*.

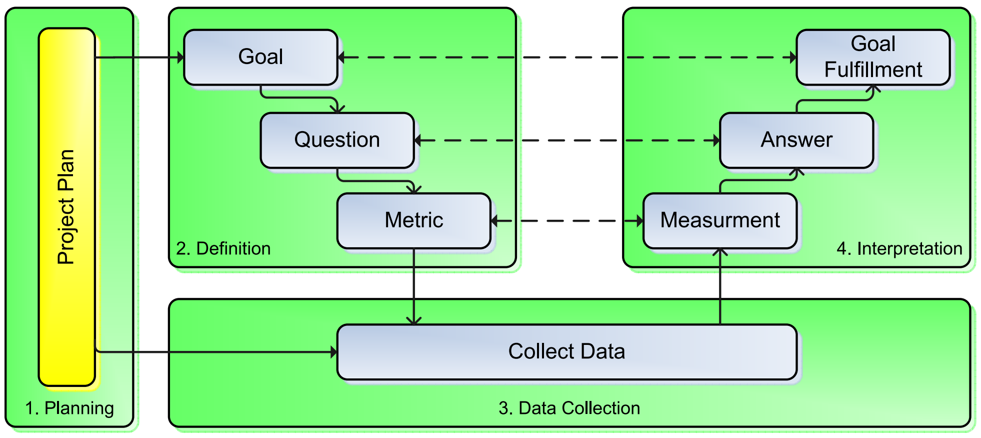


Figure 2.4: The four phases of the GQM method

The complete implementation of a GQM measurement is divided into four phases: Planning, Definition, Data collection and Interpretation [8]

2.1.5 GQM Templates

The focus of this section is to briefly introduce various templates and abstract tools which can be used while preparing and performing GQM analysis and measurements.

The relation between these techniques is illustrated in Figure 2.5.

- *Goal Template*: The Goal Template is a tool that can be used to define and characterize a GQM goal. In short, it is mainly a tool which accepts a description of goal split into dimensions as described in 2.1.1.
- *Abstraction Sheet*: The Abstraction Sheet is a tool designed mainly to support GQM interviews and to assist in defining, understanding, refactoring and reviewing a particular GQM plan. The Abstraction Sheet consists of a Goal Template and four sheet components which address predefined topics related to the GQM goal.

  The Abstraction Sheet can easily be used as a reminder or organizer in interviews as it helps the interviewer to relate raised issues to one of the four generic components and reminds him to address these components specifically if they do not appear naturally in the interview.

  An other usage of the Abstraction Sheet can be as an abstracted view of

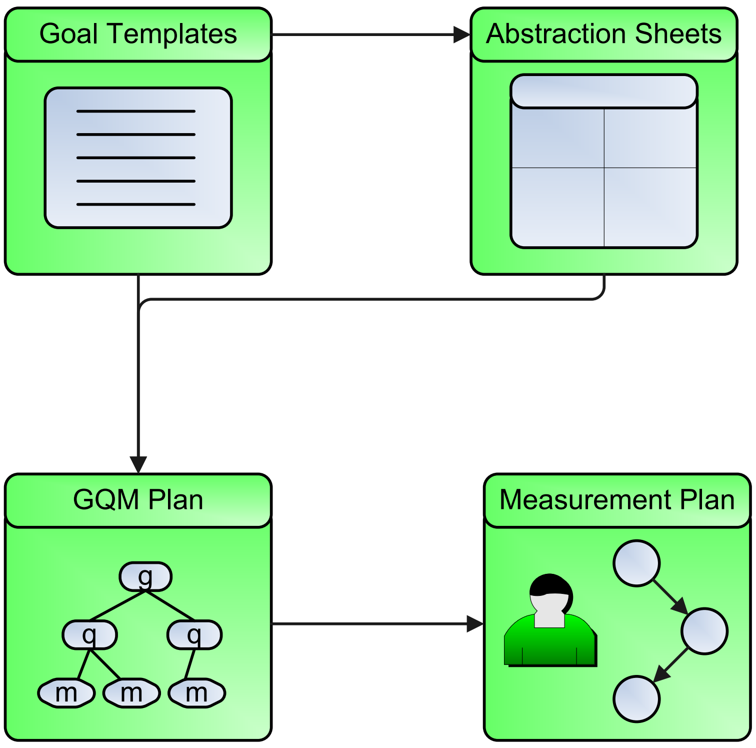


Figure 2.5: Relation between GQM techniques

The qoal template is used to create the abstraction sheet. With the goals and the abstraction sheets, the GQM plan is developed which is the basis for the measurement plan.

a GQM goal for supporting review, refactor or definition activities. The provided informations may greatly help to understand the relation of the goal between other goals and questions and the relation to the "real world" and the actual context.

The four components specifically addressed by the Abstraction Sheet are

  - *Quality Focus*: This components describes the underlying measurement *model*. It describes which aspects of the object are of interest to this goal and where these aspects are defined. This can be understood as the *type* of data to be collected.
  - Baseline Hypothesis: This component describes what actual values of data for the defined quality focuses are expected at beginning of the measurement program by the person who is interviewed. If there's no existing data, this components represents an hypothesis. These values can mainly be used for validation of the models and collected data or for relating the progress made during the measurement program.
  - Variation Factors: These components lists environmental factors which alter and affect the quality focuses. This can be helpful to explain unexpected measurement results and helps to transfer the measurement program and its results to different environments.
  - Impact on Baseline Hypothesis: Here, the anticipated impact of variation factors on the quality focuses is described. If it is not possible to describe the effects, then this strongly indicates that either the quality focuses or impact factors are setup up in a invalid manner and need rework or should be excluded.

- *GQM Plan*: A single goal plus the sets of questions and metrics that provide an operational definition of that goal. A GQM plan documents the refinement of a precisely specified measurement goal via a set of questions into a set of metrics. Thus, a GQM plan documents which metrics are used to achieve a measurement goal and explains their choice - the questions provide the rationale underlying the selection of the metrics. On the other hand, the GQM plan is used to guide analysis tasks because it documents for which purpose the respective data was collected. [21]

- *Measurement Plan*: When coupled with a GQM plan, a measurement plan specifies who collects the data required by the GQM plan, how the data is collected, and up to when the data must be collected. A measurement plan usually includes the data-collection forms as well as descriptions of tools that perform online data collection [21].

The templates introduced in this sections can either be used manually or can be integrated into a computer based tool. .

2.1.6 Actual Tool Support

Multiple reports covering problems arising during the enactment of GQM based measurement programs emerged in the last years. Many of these reports relate to the basic usage of the GQM paradigm as a "pen and paper" approach and conclude with proposing new tool implementations.

A frequently referenced problem is the high *costs* for data collection and interpretation during manual measurement and analysis cycles. These high costs restrict the measurement sample sizes, frequency of analysis, duration and the agility and flexibility of the measurement program.

These costs could be reduced significantly by automated measurement and data analysis. Reduction of the effort needed for performing measurement programs is the issue of tools like VTT Metriflame [24] or the CEFRIEL GQM tool [25].

HackyCGQM, the adaption of GQM for Hackystat developed in this work also will concentrate on reducing the measurement effort.

Today's available tools can be classified by the phases they support (overview over phases see 2.1.4). In the following enumeration, there are three examples for tools which support one, two and three phases (summarized in table 2.1).

1. GQM Planner 2 [26]
   The GQM planner provides support for the *Definition phase* of the GQM implementation. Is can be used define GQM plans and reusing parts of existing plans and enforces a very formal definition of GQM plans.
2. CEFRIEL GQM Tool [25]
   The CEFRIEL GQM Tool can be used to support the *Definition* and *Data*

tool	**Phase 1** Planning	**Phase 2** Definition	**Phase 3** Data collection	**Phase 4** Interpretation
GQM Planner 2		good support		
CEFRIEL GQM Tool		good support	manual support	
VTT Metrifame		good support	manual support	limited support

Table 2.1: GQM tools and their support

Collection phase. For supporting the data collection, it is able to plug-in itself into existing project databases, SCM tools and using Resource Standard Metrics tools generating measurements for common metrics like SLOC or CCC.

3. VTT Metriflame [24]
   Metriflame tries to support the *Definition*, *Data Collection* and *Interpretation phase* of the GQM implementation. Data collection is done by plugging into existing data bases and text files. After the data collection, functions can be defined for answering questions (most commonly by displaying graphs).

These tools provide interesting aspects, but they are still limited in respect to the degree of automation provided for measurement for the required metrics. Supporting the *Definition phase* of GQM is implemented in all of these tools, but the automated support for *Data collection* and *Interpretation* is still limited to simple GQM plans in special domains (summarized in 2.1).

2.2 Software Quality and QIP

This section introduces QIP [14], an approach for software quality improvement based on cooperative learning. cGQM, as an extended and automated GQM implementation focusing on repeated measurement cycles, can be used in an especially advantageous way in context of QIP.

Software quality is defined as all characteristics of a product that bear on its ability to satisfy explicit and implicit needs of the user [27]. As an example, the characteristics identified by International Standard Organization (ISO) are illustrated in Figure 2.6.

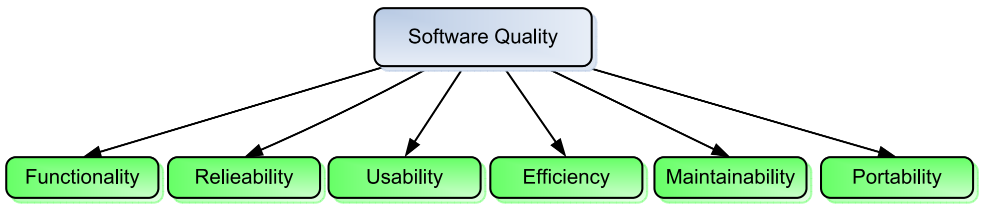


Figure 2.6: ISO 9126 software quality characteristics[27]
ISO 9126 defines six characteristic facets of software quality

Quality is an important aspect of attracting and satisfying customers. Unfortunately, the perception of quality will be different for different individuals. This makes quality a difficult, but important aspect of improvement and control.

Formal product quality is a concept that was introduced into engineering disciplines comparable late (mainly in the 1980s). After it's emersion, the impact on participating industrial sectors was dramatic. Unfortunately, software industry was one of the last sectors benefiting of this movement. Still, it's implementations are discussed and not self-evidently used.

Different paradigms dealing with software quality have been created. They can be divided into three major categories:

1. Software Improvement paradigms: These paradigms, specially designed for the software industry, aim to improve quality by improvement and learning in iterations. QIP is member of this category.
2. Software Benchmark paradigms: These paradigms try to improve software quality by setting an external scale based on known best practices. An example for this category is CMM (Capability Maturity Model).
3. Engineering Paradigms: These paradigms origin from other engineering disciplines like manufacturing. When the field of software engineering was still very young, the approach of adopting proved paradigms from other disciplines was popular. Unfortunately, the success of these adoptions was most of the time not very high [14]. Examples of this category are TQM (Total Quality Management) or Shewart-Deming PDCA (Plan Do Check Act).

The main concept of QIP is supporting the the acquisition of core competencies and improving strategic capabilities of software cooperations.

This is realized by the idea that every software project is an experiment in the sense that every project is unique and its outcome is hard to predict reliably in advance. Furthermore each project is an opportunity for the organization to learn about its processes, the products and quality aspects and to build or refine models for these objects [28].
Therefore, QIP implements two feedback and learning cycles, a cooperate feedback cycle and a project feedback cycle.

The complete QIP learning processes consist of the steps described in the following and in Figure 2.7):

1. Characterize and understand: In this phase, the current project and its environment is characterized. This supports and simplifies the knowledge transfer from already learned experience by providing the ability to compare the contexts of the actual project, and the experience for deciding if the experience is valid for the actual project or not.

2. Set goals: Set *quantifiable* goals for project and corporate success and improvements.

3. Choose models (processes, methods, techniques, tools):

4. Execute the project: While executing the project, a repeated project learning loop takes place. During the execution, the projects product is created and this process is constantly measured in in real time for applying corrective and controlling actions. This project learning process can be divided into three subphases, each representing one project feedback-loop:

   (a) Execute: Work on the product and measure that process.

   (b) Analyze Results: Interpret the measurement results for corrective actions.

   (c) Feedback: Control and correction of the project course based on the taken measurements.

5. Analyze: Analyze the projects results to evaluate current practices, determine problems, record findings and recommend improvements for future project.

6. Package: Package the experience in the form of updated and refined models and save it in an experience base to be reused on future projects.

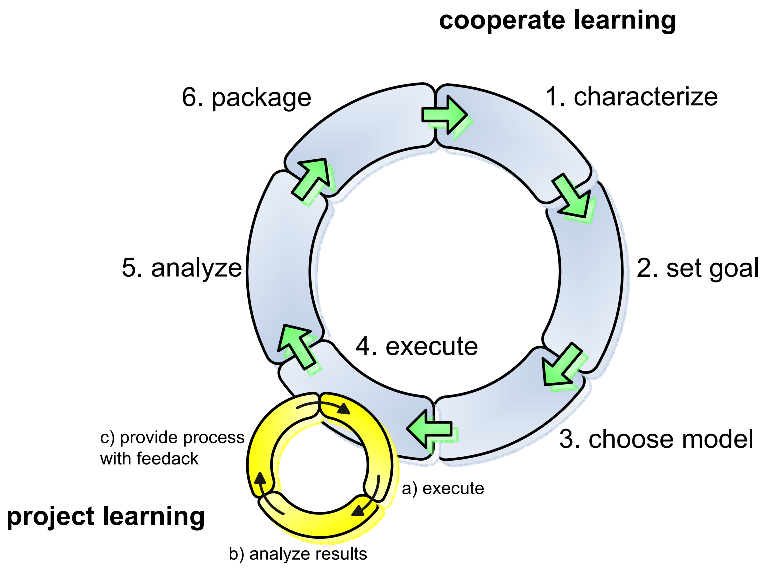


Figure 2.7: The QIP circles
QIP consists of a cooperate and a project learning circle. Both take advantage of collected metric measurements.

In the context of the QIP, GQM-based measurement is due to it's goal-based approach an ideal mechanism to support the operational definition of project goals during the planning phase, the analysis of measurement data and its feedback into the ongoing project and to allow the explicit capturing of measurement plans for reuse in future projects.

2.3 SPCC

A SPCC is defined as a platform used for controlling software development using measurement data [29].

SPCCs propose a software development model as illustrated in Figure 2.8. There, the SPCC serves as a central information processing and visualization unit providing valuable feedback data for manifold project roles.

At the actual moment, various systems are in existence which can be considered a SPCC implementation. Also hackyCGQM, which is developed during the

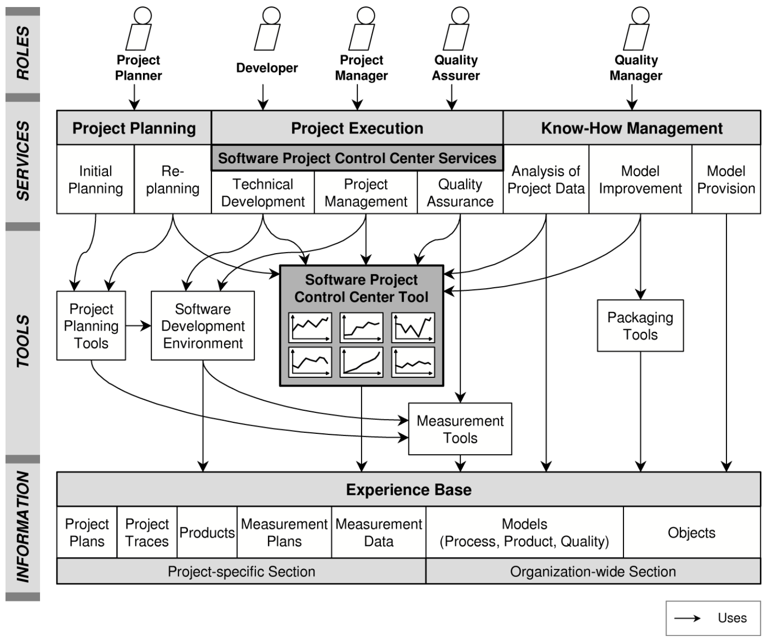


Figure 2.8: SPCCs Software Development Model (from [29])
The SPCC provides feedback for project planning, project execution and know-how management.

course of this thesis, can be considered as a specialized SPCC implementation.

The G-SPCC Reference Implementation This subsection briefly introduces the G-SPCC reference implementation [30, 31] for SPCC developed in 2003 at the University of Kaiserslautern and some of it's concepts.

A main feature of G-SPCC is that it provides *goal-oriented interpretation* and *visualization* of collected measurement data. The data interpretation is performed by so-called *visualization catenas* or *visualization chains*, which are realized by a set of combinable functions geared for purpose-oriented interpretation. SPCC visualizations are presented through a set of *role-based views*, tailored for the needs of the audience.

The G-SPCC reference implementation is realized as a web-application primary based on Java, eXtensible Meta Language (XML) and eXtensible Stylesheet Language Transformation (XSLT) on technological side.

It can access databases containing measurement data and process and visualize the data in a way suitable for project control.

The *visualization catena* and *views* are central concepts of the implementation and are explained in the following.

Visualization Catenas Data that is to be processed and visualized usually serves a specific purpose. For transforming it from it's raw form to form fulfilling this purpose, it has to be treated with different methods or techniques.

G-SPCC considers each of these methods or techniques which serve a single or multiple data transformation purposes as a "functions". Functions could, for example, predict unknown data, summarize data or calculate the deviation of a given base line.

Each function is implemented by a Java class which can be instantiated and connected to a dynamic network where the results of one function is fed into the input of the next. These networks can be defined in a flexible and complex way where functions can be fed by multiple others or can serve multiple others themselves, as illustrated in Figure 2.9.

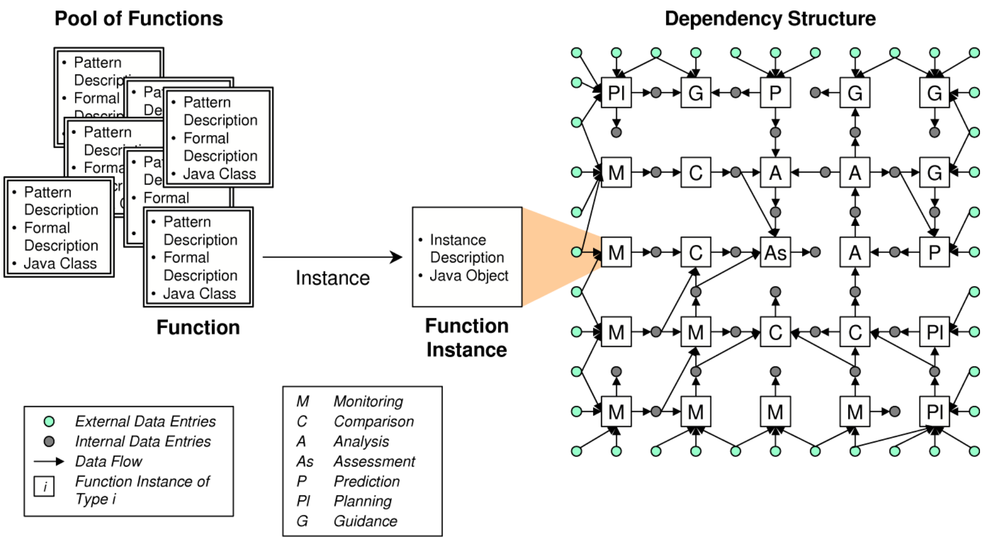


Figure 2.9: Structure of a visualization catena (from [31])
The shown catena consists of several layers of dependent SPCC function instances, which produce internal data entries.

Views SPCC suggests that the visualization of measurement data has to be role-based. This means, that for each measurement program different user roles can be defined with different requirements and needs for the visualized data. These different needs are satisfied by introducing different views at the data. Each view visualizes just the relevant data required by the role in such a way so that it satisfies the roles needs. As an example, a project manager needs different data than a developer or a business strategist.

The views are connected to the network of functions which will provide the data for the visualization. This is indicated in Figure 2.10.

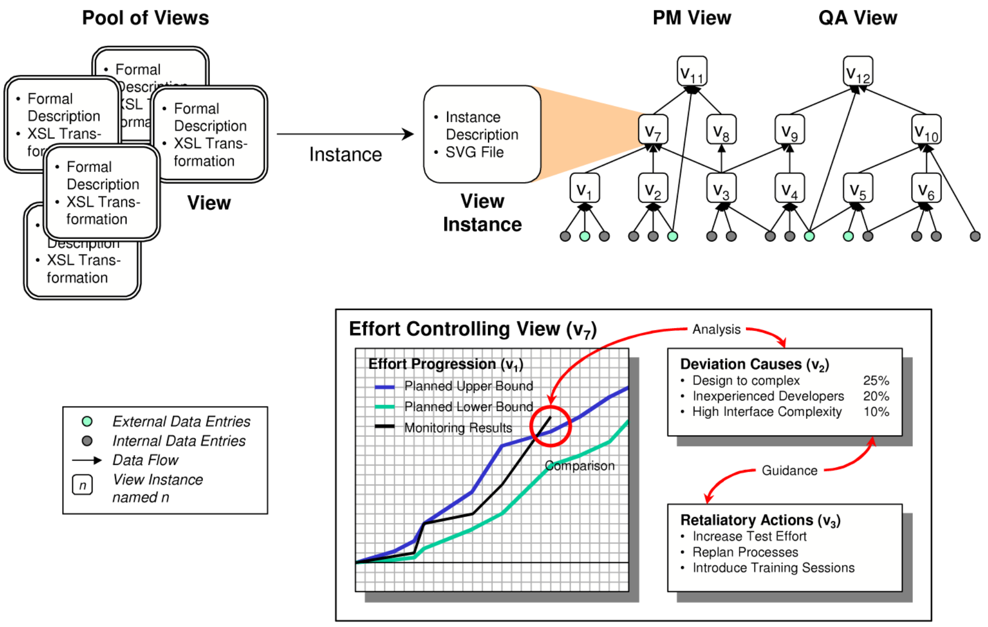


Figure 2.10: SPCC views and their instances. (from [31])
The upper part shows a pool of different SPCC views, which are instantiated to tree structures of comprising SPCC view instances suited for a project manager and quality assurer. The lower Effort Controlling View shows a sample visualization of one tree branch.

As an other example, Figure 2.11 shows a tree structured *visualization catena*. There, data is processed in layers with deeper layers feeding higher layers which finally feed the views.
This kind of tree-like design could also be used to emulate data processing in GQM scenarios (three layers: goals, questions and metrics).

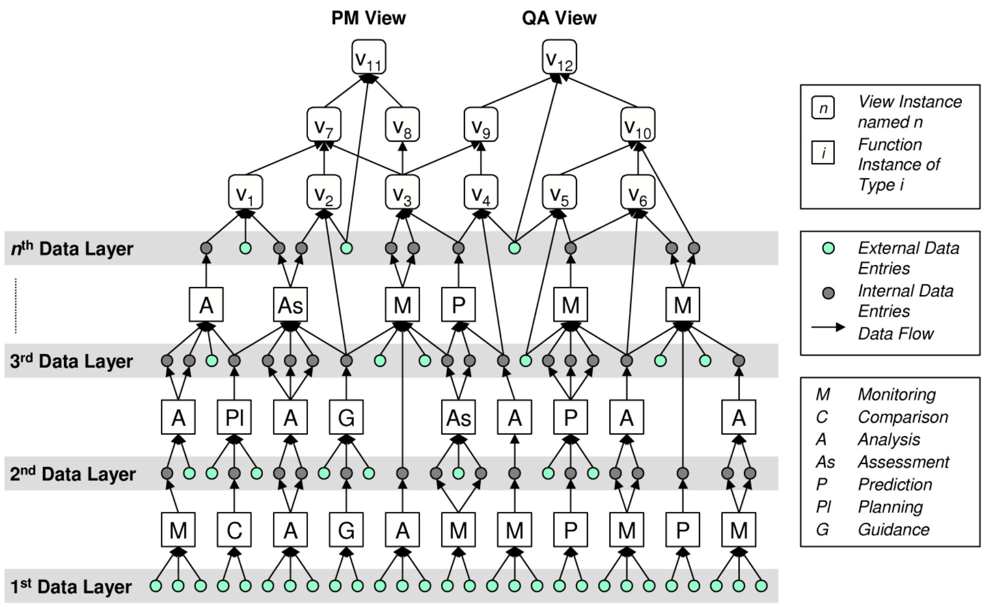


Figure 2.11: SPCC functions and their instances (from [31])
The left hand side shows a pool of different SPCC functions, which are instantiated to a complex structure of dependent SPCC function instances, shown on the right hand side.

2.4 Hackystat

This section introduces the Hackystat [1] system, an automated tool for empirically guided software process improvement. HackyCGQM, the reference implementation for cGQM proposed in this thesis, is embedded into the Hackystat system. Therefore, a brief introduction into Hackystat is needed to understand the architecture of the implementation.

Hackystat resulted from the research on PSP [32, 33, 15], the Personal Software Process. While early tools to support PSP only contained manual support (first generation tools) or semi-automated support, Hackystat is one of the first third generation tools which provides fully automated support for a wide range of software metric collection and analysis. First and second generation system were used to verify the underlying PSP assumption that there were easily recognizable relationships between internal attributes of development (i.e. directly measurable attributes such as size and time) and external attributes of development (non-directly measurable attributes such as quality and dependability). Hackystat is tailored to be a framework that allows individuals to collect data regarding their development practices in order to experimentally determine whether or not these relationships exist. In other words, Hackystat focuses on measurement validation as an important precursor to measurement application.

Measurement validation is traditionally a time-consuming and expensive process, and to make it practical on an individual level, Hackystat makes both data collection and analysis entirely automatic. Software "sensors" are attached to the developer's software tools and automatically send process data to a centralized web server. The next subsection while describe this architecture and function of Hackystat in further detail.

2.4.1 Hackystat Architecture

This subsection provides a brief overview over the Hackystat architecture and it's micro kernel. This information is helpful for understanding the architecture and function of the hackyCGQM implementation which is introduced in the later chapters.

One way of describing Hackystat's architecture is as a client server system as illustrated in Figure 2.12.

In this view, the "clients" are development environment tools, such as editors, configuration management systems, build tools, unit testing tools, and so forth. For each of these tools, a custom Hackystat sensor must be developed. It is "custom" in the sense that it must use the plug-in or extension point API for the tool, and "custom" in the sense that the type of product or process data that it collects is specific to the tool it supports.

Once data is collected by these client-side sensors, it is transmitted using Simple Object Access Protocol (SOAP) [34] to the "server", which is a web application running within a conventional servlet-supporting web server such as Apache Tomcat [35]. The client-side sensors have the ability to cache data in the event that a network connection cannot be made to the server and resend it later, allowing the developers to work offline.

Upon receipt of the "raw" sensor data by the server, various analyses can be run. Some of these analyses are run automatically by the server each day, others are run only when invoked by developers from a Web Browser interface. The goal of these analyses are typically to create abstractions of the raw sensor data stream that help developers and managers to interpret the current state and trajectory of the project and gain insight into possible problems or opportunities for improvement going forward.

In certain cases, these abstractions can be automatically emailed back to the developers on a daily basis, creating a feedback loop.

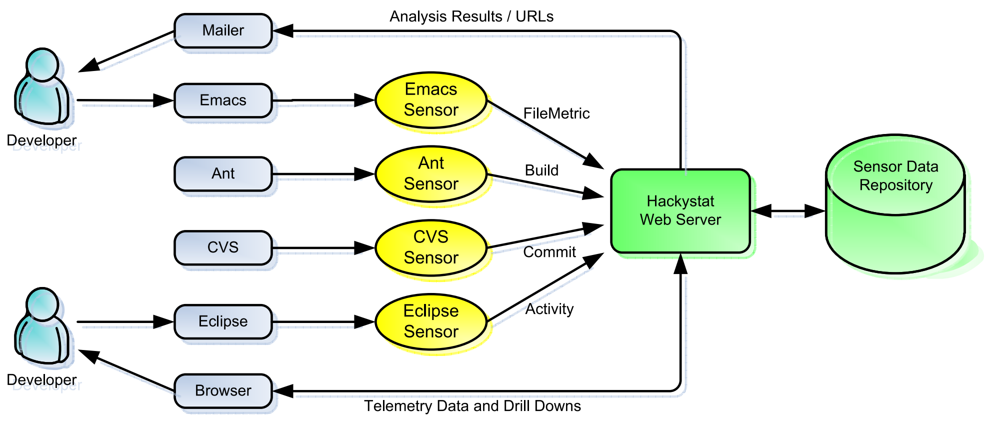


Figure 2.12: General Hacktystat architecture

Users can use different tools with sensors attached. These sensors will unintrusivly collect data and send it to the Hackystat server.

Hackystat features a micro-kernel architecture as illustrated in Figure 2.13. The kernel only provides essential services, such as receiving sensor data and managing their persistence. All additional services as analysis and alerts are implemented through through pluggable extensions. Thus an easily extensible and flexible framework is provided. This feature is used to integrate hackyCGQM, the sample implementation of this thesis, into Hackystat.

2.4.2 Metrics Collection

Hackystat supports the collection of various metrics. These can be divided into product metrics (like size, complexity, dependencies, etc) and process metrics (like active time, build related metrics, trouble-ticket related metrics) (see 1.1). Each of the supported metrics can be attached to various sensors which will measure actual metric data. These sensors are usually tailored to be integrated

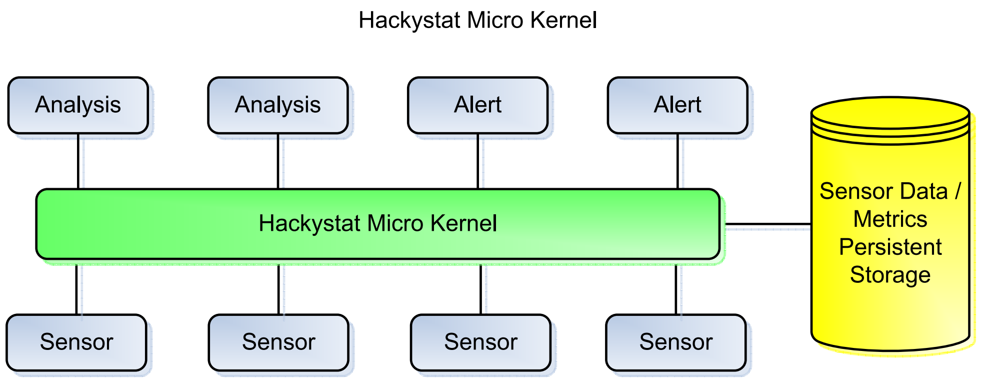


Figure 2.13: The Hackystat microkernel

The micro kernel provides a dynamic extension architecyure which allows to plug-in new modules providing new sensor types, analysis or alerts.

into a specific development tool. All metric data collected by sensors will be send to the Hackystat server it is are stored and processed.

Sensors communicate with the Hackystat server using SOAP through HTTP channels. At the actual time, sensors are implemented for interactive development environments (such as Eclipse, Emacs, JBuilder, Vim, and Visual Studio), office productivity applications(such as Word, Excel, and Powerpoint), size measurement tools (such as CCCC and LOCC), testing tools (such as JUnit and JBlanket), configuration management tools (like CVS and Subversion) and defect tracking tools (such as Jira).

Table 2.2 shows an overview over the current metrics collected by Hackystat with their respective tools. This list changes frequently as new sensors or metrics are developed and added (list below was actual due to June 31th 2005).

metric	**submetric**	**sensor**
effort	active time	Eclipse
		Emacs
		JBuilder
		MS Office
		Vim
		Visual Studio
	review time	Jupiter
filemetrics	Size	BCML

metric	**submetric**	**sensor**
		CCCC LOCC
	OO metrics	BCML CCCC
complexity	call dependency	DependencyFinder
testing	unit test results	JUnit Eclipse CPPUnit
	coverage	JBlanket
defects	trouble tickets	JIRA
	review defects	Jupiter
build	build invocation	Ant Eclipse
versioning	commits	CVS Subversion
profiling	load testing	hackyLoadTest
	paralell computing	hackyHPC

Table 2.2: Metrics collected by Hackystat

2.4.3 Analyses and Alerts

After sensor data is received at the server, it is stored and available for analysis.

Hackystat supports two styles of information retrieval: *pull* and *push*. Pull-style information retrieval is called "analysis". Developers log onto the Hackystat server and run analyses on their product and process metrics through a web interface. Push-style information retrieval is called "alert". Hackystat sends out automated email message when incoming metrics activate user-defined triggers. The message usually contains summary information, and a link which directs users to the Hackystat server for more detailed pull-style analysis.

cGQM will integrate into this concept by defining a new "pull"-style analysis, plugged into the micro kernel architecture. At the actual moment, Hackystat does not provide goal-oriented analysis; integrating cGQM would greatly improve it's abilities.

2.4.4 HackyTelemetry

hackyTelemetry is a analysis module of Hackystat. It is based on software project telemetry[36], a methodology for in-process, empirically-guided software development process problem detection and diagnosis. Metrics are abstracted in real time to telemetry streams, charts, and reports,which represent high-level perspectives on software development. Telemetry trends recognizable in these charts can be the basis for decision-making in project management and process improvement. Unlike other approaches which are primarily based on historical project database and focused on comparison of similar projects, software project telemetry focuses on project dynamics and in-process control. It tries to combines both precision of traditional project management techniques and flexibility promoted by agile community.

hackyTelemetry tries to realize these claims by plotting telemetry charts. These telemetry charts are basically line charts visualizing the development of selected data aspects over a given time span. A sample telemetry chart is displayed in Figure 2.14. hackyTelemetry is backed by a flexible language framework, the "Hackystat Telemetry Language".

Based on it's properties and abilities, hackyTelemetry can also be considered as an SPCC implementation (see 2.3).

2.4.5 Using Hackystat

As hackyCGQM, the product developed during this thesis, fully integrates into Hackystat, it is important that the reader understands how it feels to work with Hackystat. This subsection tries to help to increase this understanding using screenshots of the running system.

HackyCGQM completely relies on the data collected by Hackystat. This data is collected using various sensors installed on the developers machines as described in the previous sections.

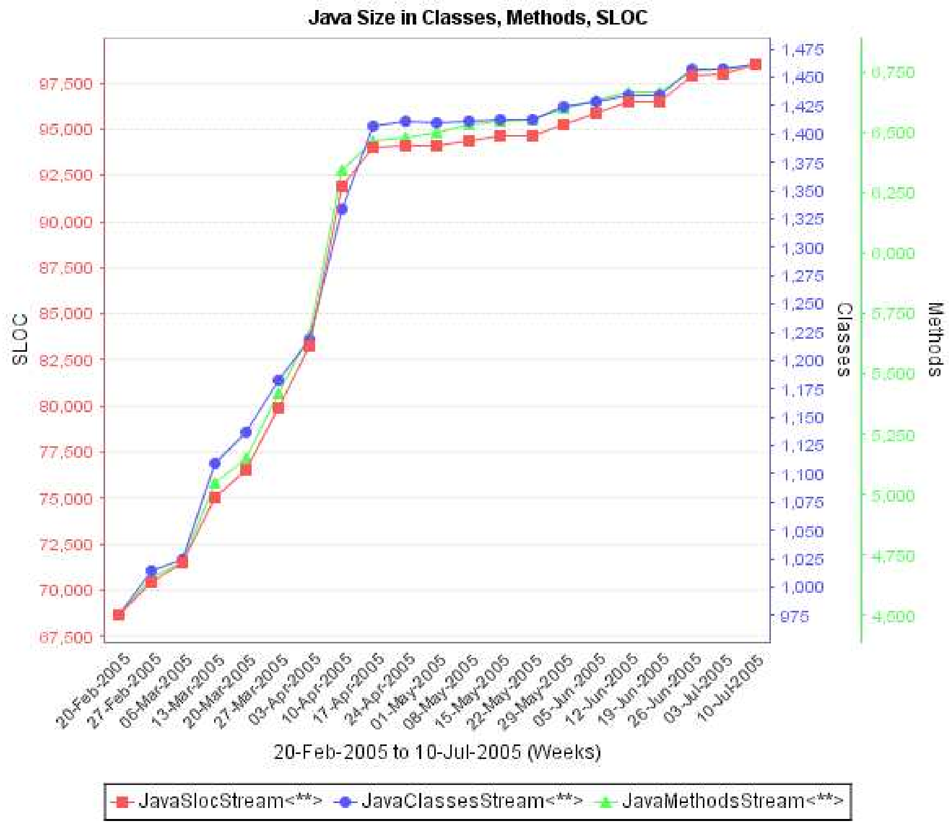
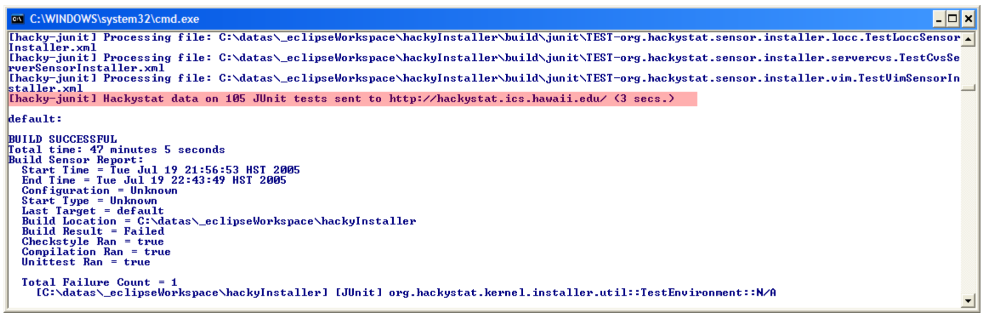


Figure 2.14: A telemetry chart

This sample telemetry chart plots the dynamics of project size change for a given time span.

All these sensors work unobtrusively in the background of the developers tools. Screenshots of this setting are displayed in Figure 2.15 and Figure 2.16 where the interaction of the Ant sensor and the Eclipse sensor with their respective host tools is displayed.

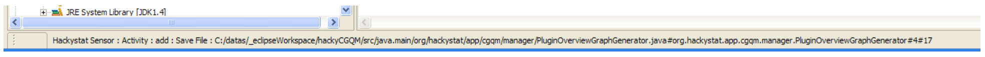
Usually, developers do not even realize the presence of the sensors as the small status messages are the only indication of their work.



```
[hacky-junit] Processing file: C:\datas\_eclipseWorkspace\hackyInstaller\build\junit\TEST-org.hackystat.sensor.installer.locc.TestLoccSensor
Installer.xml
[hacky-junit] Processing file: C:\datas\_eclipseWorkspace\hackyInstaller\build\junit\TEST-org.hackystat.sensor.installer.servercvs.TestCvsSe
rverSensorInstaller.xml
[hacky-junit] Processing file: C:\datas\_eclipseWorkspace\hackyInstaller\build\junit\TEST-org.hackystat.sensor.installer.vim.TestVimSensorIn
staller.xml
[hacky-junit] Hackystat data on 105 JUnit tests sent to http://hackystat.ics.hawaii.edu/ (3 secs.)

default:

BUILD SUCCESSFUL
Total time: 47 minutes 5 seconds
Build Sensor Report:
  Start Time = Tue Jul 19 21:56:53 HST 2005
  End Time = Tue Jul 19 22:43:49 HST 2005
  Configuration = Unknown
  Start Type = Unknown
  Last Target = default
  Build Location = C:\datas\_eclipseWorkspace\hackyInstaller
  Build Result = Failed
  Checkstyle Ran = true
  Compilation Ran = true
  Unittest Ran = true

  Total Failure Count = 1
    [C:\datas\_eclipseWorkspace\hackyInstaller] [JUnit] org.hackystat.kernel.installer.util::TestEnvironment::N/A
```

Figure 2.15: Screenshot of the Ant sensor

Hackystat provides a sensor which unobtrusively integrates into various development tools. This screenshot shows the execution of an Ant script. The ant sensor just "hooks" into the execution, no manual effort is needed to execute it.



Hackystat Sensor : Activity : add : Save File : C:/datas/_eclipseWorkspace/hackyCGQM/src/java.main/org/hackystat/app/cgqm/manager/PluginOverviewGraphGenerator.java#org.hackystat.app.cgqm.manager.PluginOverviewGraphGenerator#4#17

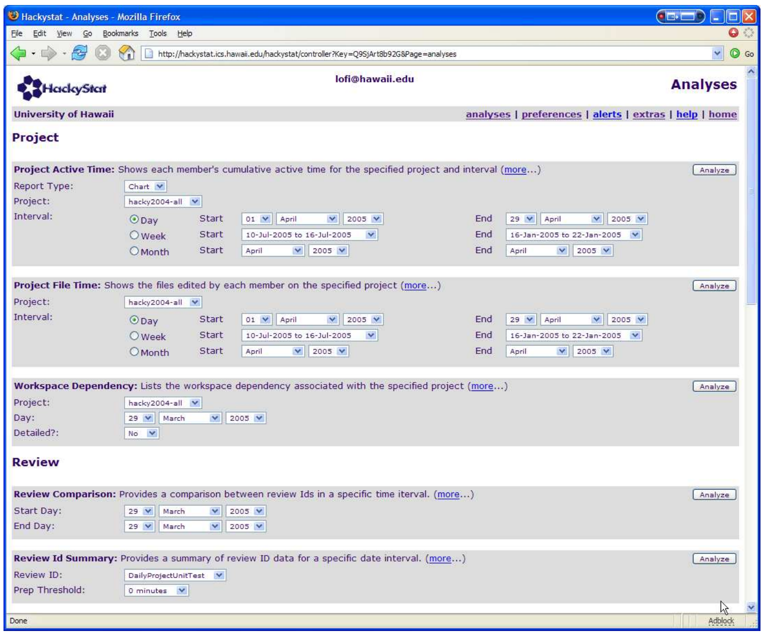
Figure 2.16: Screenshot of the Eclipse sensor

The eclipse sensor is the most popular sensor Hackystat provides. It measures various metrics like activity, unit testing, or build invocations and runs as a background task in Eclipse. The sensor is nearly unobtrusive, the shown status message is the only indictation of it's presence.

All sensor data is collected into the Hackystat data repository and available for analysis.

Usually, analysis are accessed using the Hackystat web interface as illustrated in Figure 2.17. Currently, Hackystat provides 22 different analysis, hackyCGQM is one of them.

Other examples of popular analysis are telemetry [36] displayed in Figure 2.18 and "Daily Diary", displayed in Figure 2.19.



Hackystat - Analyses - Mozilla Firefox

http://hackystat.ics.hawaii.edu/hackystat/controller?Key=Q9SjArt8b92G&Page=analyses

lofi@hawaii.edu

Analyses

University of Hawaii

analyses | preferences | alerts | extras | help | home

Project

Project Active Time: Shows each member's cumulative active time for the specified project and interval (more...)

Report Type: Chart

Project: hacky2004-all

Interval:

[✓ Day]	Start	01 April 2005	End	29 April 2005
Week	Start	10-Jul-2005 to 16-Jul-2005	End	16-Jan-2005 to 22-Jan-2005
Month	Start	April 2005	End	April 2005

Project File Time: Shows the files edited by each member on the specified project (more...)

Project: hacky2004-all

Interval:

[✓ Day]	Start	01 April 2005	End	29 April 2005
Week	Start	10-Jul-2005 to 16-Jul-2005	End	16-Jan-2005 to 22-Jan-2005
Month	Start	April 2005	End	April 2005

Workspace Dependency: Lists the workspace dependency associated with the specified project (more...)

Project: hacky2004-all

Day: 29 March 2005

Detailed?: No

Review

Review Comparison: Provides a comparison between review Ids in a specific time iterval. (more...)

Start Day: 29 March 2005

End Day: 29 March 2005

Review Id Summary: Provides a summary of review ID data for a specific date interval. (more...)

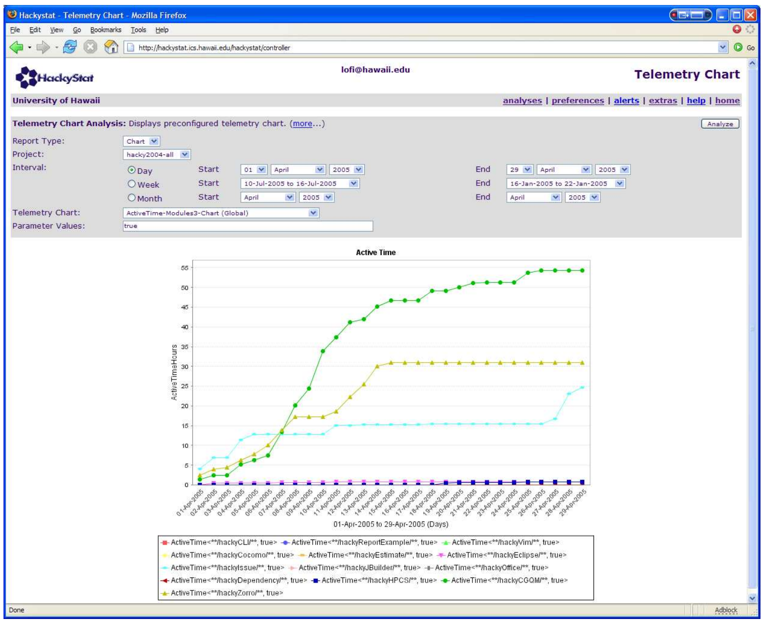
Review ID: DailyProjectUnitTest

Prep Threshold: 0 minutes

Done

Figure 2.17: Screenshot of a telemetry chart

Telemetry is a flexible analyis framework on top of Hackystat which shows trends of measurements over time



Hackystat - Telemetry Chart - Mozilla Firefox

http://hackystat.ics.hawaii.edu/hackystat/controller

lofi@hawaii.edu

Telemetry Chart

University of Hawaii

analyses | preferences | alerts | extras | help | home

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: hacky2004-all

Interval:

[✓ Day]	Start	01 April 2005	End	29 April 2005
Week	Start	10-Jul-2005 to 16-Jul-2005	End	16-Jan-2005 to 22-Jan-2005
Month	Start	April 2005	End	April 2005

Telemetry Chart: ActiveTime-Modules3-Chart (Global)

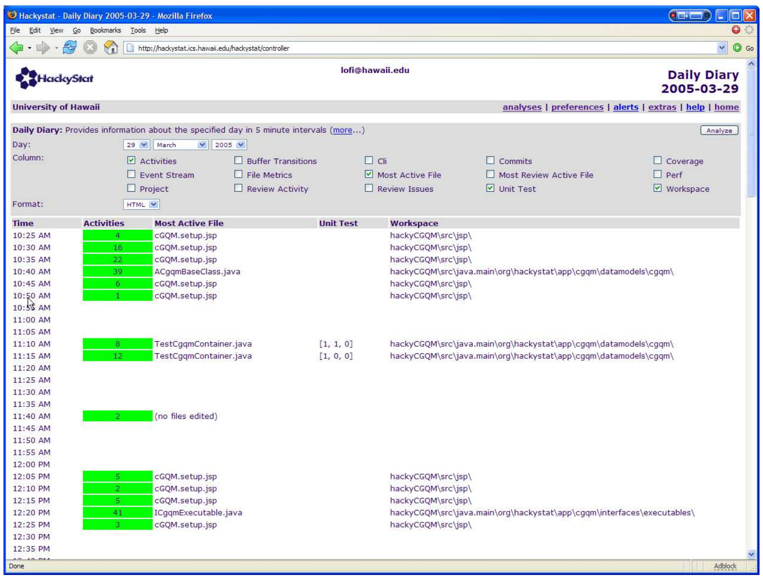
Parameter Values: true

Active Time
ActiveTimeHours
0
5
10
15
20
25
30
35
40
45
50
55
01-Apr-2005
29-Apr-2005
01-Apr-2005 to 29-Apr-2005 (Days)
ActiveTime<**/hackyCLI/**, true>
ActiveTime<**/hackyReportExample/**, true>
ActiveTime<**/hackyVim/**, true>
ActiveTime<**/hackyCocomo/**, true>
ActiveTime<**/hackyEstimate/**, true>
ActiveTime<**/hackyEclipse/**, true>
ActiveTime<**/hackyIssue/**, true>
ActiveTime<**/hackyJBuilder/**, true>
ActiveTime<**/hackyOffice/**, true>
ActiveTime<**/hackyDependency/**, true>
ActiveTime<**/hackyHPCS/**, true>
ActiveTime<**/hackyCGQM/**, true>
ActiveTime<**/hackyZorro/**, true>

Done

Figure 2.18: Screenshot of a telemetry chart

Telemetry is a flexible analyis framework on top of Hackystat which shows trends of measurements over time


Hackystat - Daily Diary 2005-03-29 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Go
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done
Adblock

lofi@hawaii.edu

Daily Diary
2005-03-29

University of Hawaii

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

Day: 29 March 2005

Column: [✓ Activities] Buffer Transitions, Cli, Commits, Coverage, Event Stream, File Metrics, [✓ Most Active File] Most Review Active File, Perf, Project, Review Activity, Review Issues, [✓ Unit Test] [✓ Workspace]

Format: HTML

Time	Activities	Most Active File	Unit Test	Workspace
10:25 AM	4	cGQM.setup.jsp		hackyCGQM\src\jsp\
10:30 AM	16	cGQM.setup.jsp		hackyCGQM\src\jsp\
10:35 AM	22	cGQM.setup.jsp		hackyCGQM\src\jsp\
10:40 AM	39	ACgqmBaseClass.java		hackyCGQM\src\java.main\org\hackystat\app\cgqm\datamodels\cgqm\
10:45 AM	6	cGQM.setup.jsp		hackyCGQM\src\jsp\
10:50 AM	1	cGQM.setup.jsp		hackyCGQM\src\jsp\
10:55 AM				
11:00 AM				
11:05 AM				
11:10 AM	8	TestCgqmContainer.java	[1, 1, 0]	hackyCGQM\src\java.main\org\hackystat\app\cgqm\datamodels\cgqm\
11:15 AM	12	TestCgqmContainer.java	[1, 0, 0]	hackyCGQM\src\java.main\org\hackystat\app\cgqm\datamodels\cgqm\
11:20 AM				
11:25 AM				
11:30 AM				
11:35 AM				
11:40 AM	2	(no files edited)		
11:45 AM				
11:50 AM				
11:55 AM				
12:00 PM				
12:05 PM	5	cGQM.setup.jsp		hackyCGQM\src\jsp\
12:10 PM	2	cGQM.setup.jsp		hackyCGQM\src\jsp\
12:15 PM	5	cGQM.setup.jsp		hackyCGQM\src\jsp\
12:20 PM	41	ICgqmExecutable.java		hackyCGQM\src\java.main\org\hackystat\app\cgqm\interfaces\executables\
12:25 PM	3	cGQM.setup.jsp		hackyCGQM\src\jsp\
12:30 PM				
12:35 PM				

Figure 2.19: Screenshot of the DailyDiary analyis

The daily diary is a low level abstraction of measured data for one developer and one project where events and data is plotted over the course of one day.

Chapter 3

Continous GQM

Hackystat (see 2.4) is a fully automated measurement system. Therefore, adding support for the GQM Paradigm as mentioned in the introduction will allow the execution of automated goal-oriented measurement programs. These fully automated measurement programs will have specific properties and abilities distinguishing them from manual programs, but are limited to the usage of metrics which are able to be measured automatically.

The concept of this automated, goal-oriented GQM based measurement will be called 'continuous GQM' (cGQM) in the following, inspired by it's ability of continuous measurement due to shortened measurement cycles.

This chapter will illustrate the theoretical foundations of cGQM as a concept while chapter 4 will highlight *hackyCGQM*, the reference implementation of the cGQM concept which is integrated into Hackystat.

One of the major aspects of cGQM, as it's name proposes, is *continuous measurement* as described in the next section.

3.1 Continuous Measurement

The expression *continuous measurement* in the context of cGQM refers to measurement programs with extremly short feedback cycles. This means that up-to-date metrics, question answers and goal fulfillment degrees from the underlying GQM plans are accessible any given time. This ability enables the fine grained tracking of the measurement and analysis results over a longer period of time.

The availability of *actual* measurement and analysis data at any given time opens new ways of data collection and analysis, as changes and trends depending on *time* can be identified and visualized and is thus superior to measurement programs which include just a single or only few, discreet measurement cycles.

For being able to perform continuous measurement in the context of GQM efficiently, following prerequisites have to be fulfilled:

1. Ability to track, store, visualize and analyze the development of GQM results over time:

   Fine-grained analysis of trends and ability to track time based behavior of GQM goals and questions is one of the primary advantages which is gained by continuous measurement. Tools and visualizations to support this have to be available to claim the maximum positive effect of this.

2. Reduction of the costs of an actual measurement cycle as far as possible: The success of continuous measurement mainly depends on the costs which are connected to a single measurement cycle. The higher the costs are, the less attractive it is to perform an actual measurement. The optimal case would be a zero cost measurement and analysis cycle which can be executed without any connected overhead.

   In context of GQM, a measurement cycle can be broken down into the following sub-phases which can be optimized to a more cost effective level individually:

   (a) Metric data collection: In the best case, all needed data connected to the defined metrics in a GQM plan are measured automatically without any manual overhead. This is already possible for a wide range of metrics (as, for example, implemented by Hackystat, see 2.4), but still there remains a set of metrics for which there is no measurement possible in a completely automated fashion (as for example metrics connected to human behavior or opinions).

   (b) GQM analysis: After the necessary metric data is collected, it can be used for answering the questions and calculating the goal fulfillment degrees (see 3.3).

The primary goal of cGQM is to implement completely automated measurement and measurement analysis cycles enabling the usage of *continuous measurement*.

3.2 Automated Measurement

cGQM focuses only on *automated* measurement and analysis. Although this has numerous advantages like low overhead, high reliability or low costs, it is unfortunately not always possible archive full automation.

One possibility to implement automated measurement is using software tool sensors like in the Hackystat approach (see 2.4) used for the cGQM reference implementation. By doing this, a wide range of software product and process metrics can be measured, but there are also many metrics which are not measurable in such a way.

Problems arise with all metrics which are not dependent or only limited dependent on the use of software tools.

As an example for a partial metric, "project effort" is chosen: Hackystat provides a proxy metric for "project effort" called "active time". "Active time" measures all the time a developer is *actively* using his development tools. This can be done automatically using sensors. But "project effort" also should consist of time spend for meetings, planning or other activities not based on the usage of tools.

Other metrics might be even more difficult or even impossible to be measured automatically. Part of this group are all metrics based on human knowledge, estimations or assessments. According to the findings of the structural mapping evaluation in chapter 5.3, roughly around 60% of metrics used in measurement programs published in the a selction of the literature can not or only partly be measured automatically.

cGQM concentrates only on metric which can be measured automatically, limiting it's range of use.

3.3 Goal Fulfillment Degrees

cGQM introduces "goal fulfillment degrees". This means, that each goal can calculate a value representing to which degree it is fulfilled. This fulfillment degree usually should be calculated using the answers of the questions and can be understood as a summarizer for the question answers in context of the given goal.

This does not always make sense, so this concept is optional. But there are some cases, where calculated goal fulfillments are useful and can provide interesting usage scenarios.

One prerequisite for useful goal fulfillment degrees is a goal allowing the calculation of a meaningful value. Goals like "...understand what is going on..." are not suitable for this as the cGQM framework can not tell if the users did or did not understand what is going on by reading the question answers. Other goals which are clearly stated and have controllable fulfillment criteria can have meaningful fulfillment degrees. Examples of suitable goals and their fulfillment degrees could be

- "Improve unit test coverage by 10%": This is a goal with an easy to calcuate fulfillment degree. A baseline can be determined as soon as the measurement program starts. By measuring actual coverage ratios, it is easy to determine the actual progress.
- "Increase nightly integration build success rate to stable 90%": This goal is used in the "buildFailues" example in chapter 5.1.1. Here, a new fullfillment degree is calculated everyday and it represents the stability and "goodness" of the build process.

  In that case, "stable build success rate" is defined by a rated sum of the average success rates of the prior four weeks of the actual date. As soon as all four prior weeks have an success rate above 90%, the goal is considered as fulfiled.

  In this example, the fulfillment degree is represented by a graphical abstraction similar to a red light (it has 5 colors instead of 3). Everyday, project members who are not interested in all answers to all the questions, but to the measurement program in general, can just invoke the daily cGQM analysis and check the "red light", which shows them with one glimpse how well the project team is doing for fulfilling the goal.

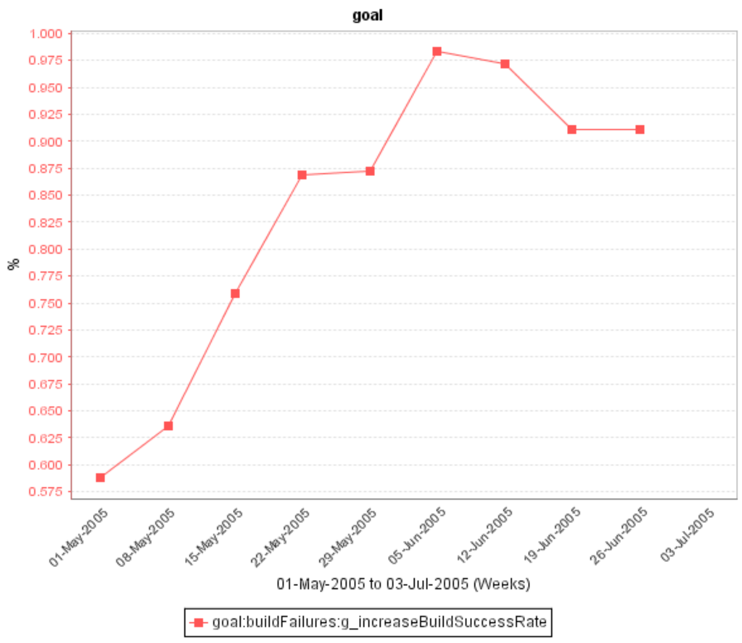


Figure 3.1: A plotted goal fulfillment degree.
This shows a numeric representation of the goal fulfillment degree for the "buildFailures" example from March to Juli. Created by the cGQM extension of hackyTelemetry.

Setting up goal fulfillment degree calculations can be difficult. In many cases, it is not possible in a meaningful way at all. In most other cases, there are many different solutions with varying difficulty. For example, in the "build success" example above, the decision was made to describe "stable" by defining a rating function and rate the prior four weeks, just using their success rate value. But there might be many different solutions, using more data or more sophisticated ratings or even being much simpler than the chosen solution.

The hard problem with fulfillment degrees is to find out which implementations are meaningful or not.

As soon as there is a meaningful implementation of a fulfillment degree calculation, it can be used as an high level abstraction for a parts of the measurement program. This abstraction can be used for continuously staying up-to-date during the course of the measurement program by checking it, for example, every day or every week ("To which degree is my goal fulfilled today?"). It can also be used for observing the trend of the measurement program over a longer period of time, as illustrated in Figure 3.1.

3.4 Calculating Goal Fulfillments

Calculating the goal fulfillment degree is not easy. As already mentioned in the last subsection, it might not even be possible in many cases. In the cases in which it is possible, the actual implementation of the fulfillment degree calculation has to be chosen with care.

Goal fulfillments in cGQM are technically calculated in the same manner than question answers (see later 4.2.2), meaning the measurement program developer can freely choose any algorithm he can think of.

The important criteria while deciding for a specific implementation is how good the calculated value *semantically* represents the fulfillment degree of the goal. So, for example, it does generally not make sense to just summarize all questions' values and average them somehow as there may be many questions which are just designed to understand the problems domain better and don't give interesting information concerning the goal fulfillment.

In cGQM, questions can be divided into two groups:

1. questions which have an *informative* character and are providing data which can be used to develop strategies for approaching the goal or understanding the problem domain. So for example, when the goal is reducing the defects in a project, a question concerning the projects' modules size might be interesting for understanding the defect distribution which can help reducing the defect density, but it does not provide data which is suitable for calculating a goal fulfillment degree. These are usually the question which are in the GQM abstraction sheet identified as having influence on the goal, but or not in the direct quality focus.

2. questions which actually represent the *status* of the actual goal fulfillment. In the above example, this could be the actual defect rate of the system

detected after shipping. Usually, these are the questions which are in the quality focus of the GQM abstraction sheet.

For calculating the goal fulfillment degree, *informative* questions of the first type can be ignored.

For actually calculating fulfillment degrees, the cGQM measurement program developer has to design an algorithm, freely implementable with the Java programming language as a *goal executable* (see later in 4.2.2). This algorithm usually will depend on answers of the type 2 questions (those describing the actual status), but is not limited to that. The developer can freely decide how to implement the calculation, ranging from simple value averaging up to the use of sophisticated artificial intelligence algorithms.

3.5 Usage Process Models

There are several ways of how to conduct a measurement program, resulting in different usage process models. Here, four usage process models are identified, defined and explained.

They are

- Consecutive process model
- Continuous process model
- Retrospective process model
- Alert-based process model

These usage models are not exclusive, switching between and mixing usage models is possible and in some cases very beneficial, as described in the follow-up section 3.6. Also, there might be additional usage models not listed here.

Table 3.1 provides a brief overview of the differences between the process models.

model	**type**	**number of measurement cycles**	**time focus**	**time between event and feedback**
consecutive	pull	one	future	long
continuous	pull	many	future	short
retrospective	pull	arbitrary	past	long
alert-based	push	many	future	very short

Table 3.1: cGQM process models compared

3.5.1 Consecutive Process Model

The consecutive process model is the cGQM base model. The model will be used as reference model for explaining the other 3 models in this section as they are just modifications of the consecutive one.

The consecutive model performs the four phases of a GQM program (see 2.1.4) in a strict, consecutive order, similar to waterfall models.

The consecutive process model can be described as an one-usage model, it consists of four phases which are executed in a linear fashion, as illustrated in Figure 3.2:

Phase 1: (*Planning phase*) In the first phase, a project for applying measurement is selected, defined, characterized and planned. This results into a project plan.

Phase 2: (*Definition phase*) Using this project plan, goals, questions and metrics are defined. Also, executables for these cGQM artifacts have to be implemented. This phase is similar to the *Planning phase* in manual GQM, but it consumes a significant higher effort as the implementation of executables can be very expensive.

Phase 3: (*Data collection phase*) After implementing the cGQM plan and it's executables, the automated data collection can be started. This should consume no manual effort in the best case.

Phase 4: (*Interpretation phase*) After all data is collected, it can be analyzed and interpreted. In cGQM, the analysis should be done as far as possible by the executables with no manual effort. But still, the final interpretation of the results has to be done by a human analyst who interprets all the generated answers, numbers and charts.

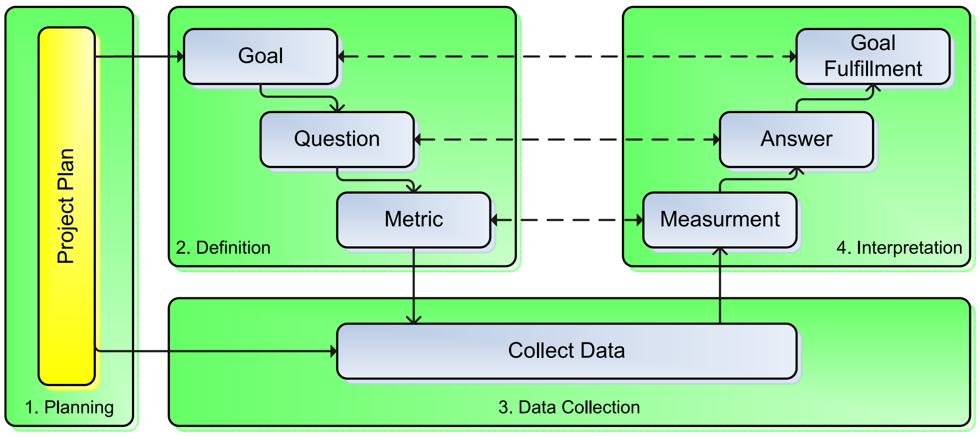


Figure 3.2: Consecutive process model[8]
This is the process model also used in consecutive GQM

The consecutive process model is usable with cGQM, but it does not take advantage of the additional abilities of cGQM compared to manually executed measurement programs with a degree as high as the three other process models in later sections do. But it still takes advantage of one of cGQM major benefits, the ability to collect and analyze huge amounts of data with minimal costs. This model can be useful for measurement programs working with "snap-shot" data, e.g. all data used can be collected on the same point in time and no longer data collection is needed.

The main disadvantage of the consecutive process is the long feedback cycle during longer data collection phases as results are only generated at the very end of the measurement program. Also, tracking of trends can be difficult as multiple measurement and interpretation cycles are needed for that. These drawbacks are solved with the *continuous process model* described in the next subsection.

3.5.2 Continuous Process Model

The *continuous process model* focuses on the shortening of the feedback cycles. The main idea of it is that intermediate analysis results should be available for no additional cost during the length of the whole measurement program after it's definition. So, the continuous model is able to *continuously collect and analyze data*.

The advantage of having arbitrary short feedback cycles is that the measurement program becomes more suitable for process controlling. Intermediate analysis results are available during the program's execution and can be used for controlling actions. Also, impacts of these controlling actions can show up in the analysis results after a very short time and can be used for further adjustments.

The continuous process is illustrated in figure Figure 3.3. Phase one and two (*planning* and *definition* phase) remain the same compared with the consecutive model. The main difference is that the *data collection* and *analysis* are executed iteratively. So for example, data can be collected automatically every day and simultaneous to that, analysis results are generated and made available.

3.5.3 Retrospective Process Model

The retrospective process model changes the order in which the cGQM phases are executed. The main change is that data and metrics are collected *before* the measurement program is designed and implemented.

This is expedient due to cGQM's ability to collect metrics without costs. In the retrospective model, *all* available metrics (those supported by the system implementing the cGQM Paradigm) are collected as soon as the project starts and are stored in a sensor data repository.

During the course of the project, cGQM based measurement programs can be implemented which can access the already collected metric data.

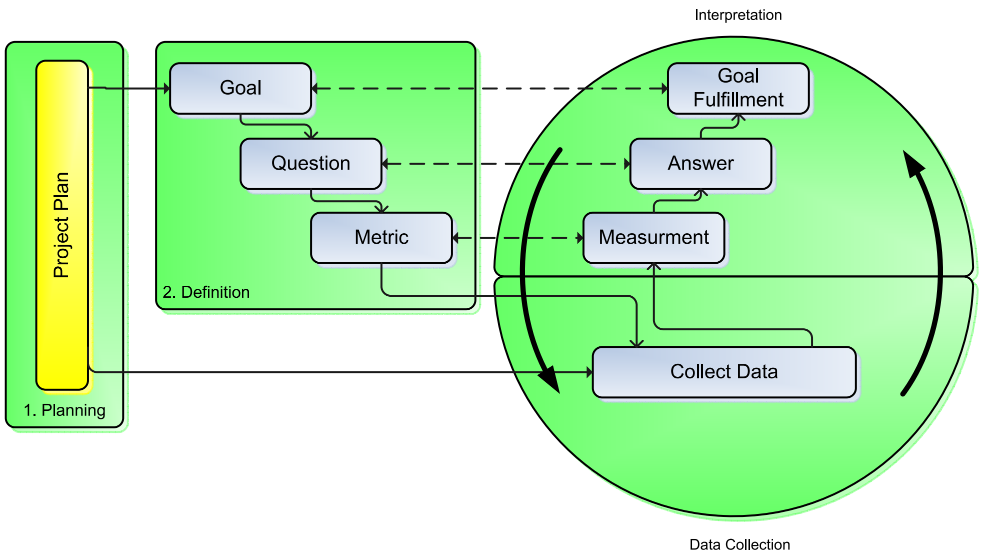


Figure 3.3: Continuous process model

The continous process model provides the ability for very short measurement and analysis cycles which can allow a continues tracking of results.

This modification contradicts one of the goals of GQM: the selection and then the sole collection of metrics which are useful and needed in the context of the measurement program. While performing this usage model, *all* possible metrics are measured. The purpose of the cGQM plan then is to determine which ones of all measured metrics should be used for analysis and interpretation.

This approach has several interesting implications:

- If during the course of the project a problem occurs which demands the implementation of a measurement program to be analyzed, the retrospective process model allows also to examine the early phases of the project before the problem was known and the measurement program implemented as all possible metric data was collected from the beginning on.
- This process model is useful when new hypotheses are developed while a measurement program result is analyzed. This is described in the next section (3.6)

It is to be noted that the application of this model does not imply significant additional costs or overhead as the collection of the metric measurements is done automatically. The only occurring overhead compared with the other process models is the storage of unnecessary data which just uses up cheap hard disk space. This claim was approved during the evaluation in chapter 5.4.

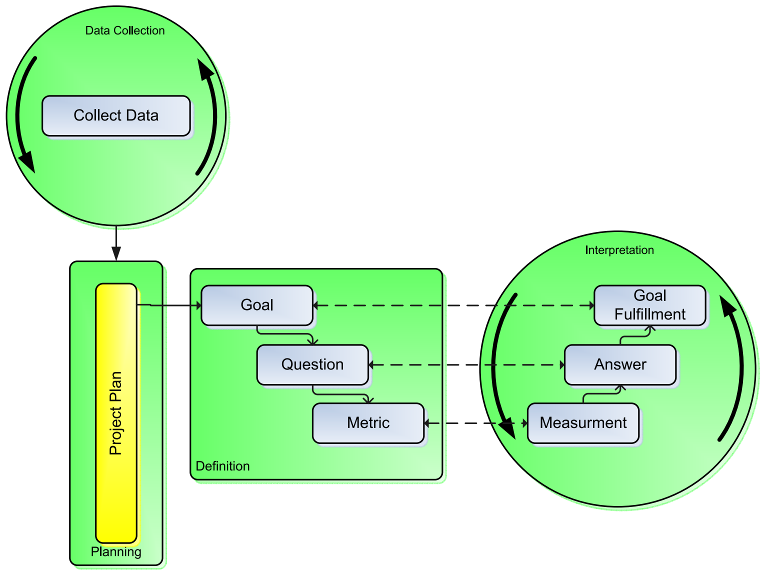


Figure 3.4: Retrospective process model

All possible metric data is collected without any specific purpose. This enables measurement programs that are implemented at later date to access data of time which is already in the past.

3.5.4 Alert-Based Process Model

This subsection introduces the alert-based process model, a passive process mostly suitable of controlling.

While the last three process models can be considered as being "active" processes (at least one person actively accesses the analysis results for final interpretation), the alert-based process model can be considered as being "passive".

"Passive" in this case means that part of the definition of the measurement program is the definition of an *alert criteria*, it's violation indicates that the observed object is "out of control".

After that, the measurement program runs completely in the background and nobody is actively paying attention to the collected data or the results of the analysis. As soon as the alert criteria is violated, the results of the last measurement cycles analysis are forwarded to the persons responsible for the measurement program. This is illustrated in Figure 3.5.

This behavior makes this process model specially suited for controlling purposes in environments where it is known how upcoming problems can be measured and identified. There, a cGQM measurement program which focuses on questions that indicate problems in a well understood process can be set up and the according alert criteria can be set.

As long as the measured process is smooth, the cGQM program stays quiet. It only reacts when problems are occurring and automatically presents analysis results helping to understand and solve the appeared problem directly to the people in charge.

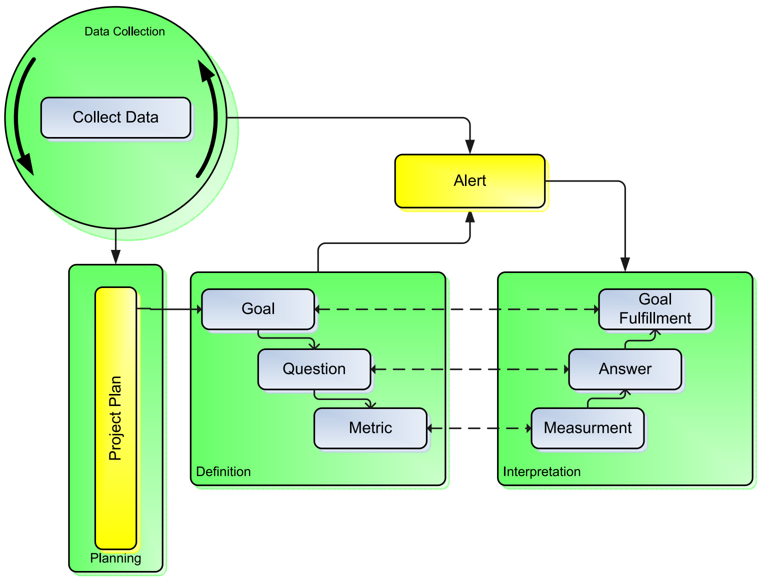


Figure 3.5: Alert-based process model

The measurement program constantly checks if a given alert criteria *is violated. As soons as this happens, it forwards the analysis results.*

3.6 Refining and Learning Process Models

This section describes the possibilities and advantages cGQM offers in respect to the ability to react and alter the program depending on it's results by combining the usage models presented in the last chapter.

While performing a measurement program, it usually provides results. These results hopefully create additional insights into the observed object or process. These insights can have multiple effects:

1. They can completely solve the problem. The measurement program ends with a complete success.
2. They can point to yet unknown problem aspects and rise new questions for understanding these new aspects.
3. They can help to develop strategies and action which are supposed to improve the observed object/process. But the effect of these actions has to be checked and confirmed.

In case two and three, it would be beneficial if the current measurement program can be modified to support the new requirements. While performing traditional GQM programs, this can be difficult as results are usually generated at the end of the program. Also, the measurement program only collects data related to it's current design. If new questions arise during or after the program's execution, it can happen that there is no data that is useful for answering the new questions.

For improved agility and flexibility, it is beneficial to mix the usage models introduced in the last chapter.

Following scenarios are valuable:

- Measurement program points out new hypotheses or questions:

  In the case of new emerging hypotheses, goals, questions or metrics, an existing cGQM plan can be just extended. After the plan was adjusted, several options for examining the impact of the newly added artifacts arise:

  1. Perform a new measurement program using the *consecutive process model*. This is the option which takes least advantage of the abilities cGQM provides.
  2. Evaluate the impact of the new artifact just for future time periods by using the *continuous process model* and concentrate on the evaluation of the new data aspects delivered by the measurement program.

3. Re-Evaluate the complete *past* run time of the measurement program using the *retrospective process model*. This is probably the most beneficial option as there is no difference between knowledge extracted from the old artifacts' data and the one extracted from the "new" artifacts. The catch of this option is that it requires the underlying data collection framework to be configured in such a way that all possibly required data is collected and being available for supporting new aspects of the measurement program.

- The measurement program was used to develop corrective or improving actions: In this case, the purpose of the measurement program changes to the task of determining if the newly implemented, corrective actions have a positive effect or not. As the measurement program was providing the data for identifying deficiencies in existing processes or products and developing corrective actions, it should also be able to show what the effects of these actions are.

  In this case, it is useful to implement the measurement program as a *longterm background* program using the *continuous process model*. By just continuing the measurement program with short feedback loops, it is possible to control and assess the corrective actions. This is, of course, also possible with any manual implementation of a GQM measurement program, but by using cGQM, it is possible to do this with *no* costs. As a conclusion of that, it is completely feasible to continue measurement programs for years, if necessary.

These abilities also support the "experimental" character of improvement programs like QIP (see 2.2) perfectly.

3.7 cGQM and Telemetry

Software telemetry [36] is a part of the Hackystat framework. The sensor data is abstracted into high-level perspectives on development trends called Telemetry Reports, which provide project members with insights useful for local, in-process decision making.

The role of hackyTelemetry and software project telemetry in context of hackyCGQM is twofold:

1. hackyTelemetry provides high level measurement data for cGQM metrics: cGQM plans can access arbitrary telemetry streams using the `TelemetryAccess` metric executable reference implementation.

2. hackyCGQM feeds data back into hackyTelemtry. This enables the dynamic and trend-oriented analysis of GQM results on the same level as all other project telemetry data: hackyCGQM provides a *telemetry reducer* for hackyTelemetry. Telemetry reducers are the building blocks of telemetry streams which will be aggregated to telemetry charts and reports. This telemetry reducer can feed the results of questions, metrics and fulfillment degrees of goals into the telemetry environment. As telemetry streams describe the development of a *numeric value* over time, it is unfortunately not possible to feed all possible cGQM artifacts into telemetry. In the actual implementation, the reducer is restricted to artifacts also returning a numeric value.

By providing cGQM analysis results for the telemetry framework, the analysis of dynamics and time-based development is simplified. It is, for example, possible to track the answer or goal-fulfillment degree of a chosen goals or question over a longer period of time and identify those points of time which yield interesting data and select those for detailed analysis. This is also described in chapter 3.3 and illustrated in Figure 3.1.

Chapter 4

HackyCGQM

HackyCGQM is the cGQM reference implementation created in the course of this work. It is integrated into Hackystat (see 2.4), a system for automated and unobtrusive metric collection. This section introduces and explains hackyCGQM.

HackyCGQM is written using Sun Java 1.4 and published under General Public License (GPL).

4.1 Key Requirements

This work does not contain a full-featured requirement analysis as this would go beyond the scope of this document. But for better understanding, the most important key requirements and their consequences are summarized in this chapter.

Requirements describe different aspects of the product to be developed and the related context. As a conclusion of that, five types of requirements or requirement related statements can be distinguished [37]:

- *Functional requirements* are the fundamental subject matter of the system and are measured by concrete means like data values, decision making logic and algorithms.
- *Non-functional requirements* are the behavioral properties that the specified functions must have, such as performance, usability, etc.

- *Project constraints* identify how the eventual product must fit into the world. For example the product might have to interface with or use some existing hardware, software or business practice, or it might have to fit within a defined budget or be ready by a defined date.
- *Project drivers* are the business- related forces. For example the purpose of the product is a project driver, as are all of the stakeholders each for different reasons.
- *Project issues* define the conditions under which the project will be done. They are included to the requirements specification to present a coherent picture of all the factors that contribute to the success or failure of the project.

In the following, the key requirements for hackyCGQM will be summarized. The presentation is inspired by the structure suggested by the Volare Requirement Templates [37]. The data contained in the templates used here is:

- Requirement Id: An unique Id.
- Requirement Type: The type of the requirement as specified in the last paragraphs.
- Description: A description of what the requirement demands.
- Rationale: A brief justification why this requirement is important.
- Consequences: The actions this requirement implies.

Requirement Id	integration
Requirement Type	project driver
Description	Hackystat has to be augmented with the ability of performing goal-oriented, GQM-based measurement plans.
Rationale	Hackystat is a powerful measurement framework suitable for serving as a primary data source for GQM like measurement programs, but lacks the ability of goal-oriented analysis. Adding a GQM extension would greatly enhance it's capabilities.

Consequences	The new extension will be called hackyCGQM and has to integrate seamlessly into Hackystat. That means, it has to be based on the same technologies and licenses (Java 1.4 programming language, restrictions on additional frameworks, predefined coding style, GPL license etc) and has to use Hackystat's framework extension points and build process.

Requirement Id	gqm
Requirement Type	project driver
Description	hackyCGQM has to support all central concepts and templates of GQM like goals, questions, metrics, abstraction sheets, goal templates or GQM plans/trees.
Rationale	People familiar with GQM should recognize all core concepts they are used to.
Consequences	Structure and naming of measurement plan definitions and presentations will be designed compatible with the GQM Paradigm.

Requirement Id	flexibility
Requirement Type	project driver
Description	New GQM plans for the hackyCGQM system should be able to be developed independently of the main system in a manner as flexible as possible.
Rationale	Measurement plans for hackyCGQM are not supposed to be hard-coded into the system as this complicates the development of new plans.
Consequences	HackyCGQM will provide an extensible framework architecture with the ability of loading plugins containing new measurement plans dynamically. Also, the development process of these plugins has to be supported.

4.2 Definition of cGQM Plans

This section briefly illustrates how the definition of cGQM plans is realized in hackyCGQM. This is necessary before describing hackyCGQM's architecture as the concepts introduced in this section have an high impact on the architecture.

For doing this, several concepts have to be introduced:

- Plugins: Executable cGQM plan including its formal descriptions and executables
- Executables: Code fragment which performs an automated analysis or metric collection.
- Plugin describers: Formal descriptions of cGQM plans, part of a plugin.
- Plugin Development Environment: Supportive environment for developing new plugins
- Plugin validation: Process of validating the syntactical correctness of plugins

4.2.1 cGQM Plugins

cGQM Plugins can be described as machine executable GQM plans. They contain a *plugin describer* describing the GQM artifacts (the term *artifacts* is used as a summarizing term for goals, questions and metrics) and their relation to each other. In addition to that, the plugin also contains *executables*. These are code fragments which will actually perform the calculations needed for answering the questions or accessing the data for the metrics.

Plugins can be deployed to a cGQM server which can execute the defined measurement program. For better reuse of already created executables, plugins can also access executables which are stored in other plugins. This ability allows the creation of cGQM libraries where in the best case a new plugin just needs a new describer, but can reuse already existing executables.

In hackyCGQM, plugins are realized as dynamically loaded `jar` files containing the *executables* and the *plugin describer*.

4.2.2 Executables

Executables are code fragments performing the calculations for cGQM artifacts (e.g. answering questions or accessing metric data). Each artifact is bound to an executable. This binding is defined in the plugin describer.

Executables in hackyCGQM are characterized by following properties:

- Interfaces: Each executable has to implement one of the according sub-interfaces of `IExecutable` provided by the hackyCGQM framework (e.g. `IGoalExecutable`, `IQuestionExecutable` or `IMetricExecutable`). These interfaces include methods for executing the executable and setting and validating parameters and slots.
- Parameters: Each executable can have a set of parameters. Parameters are named value pairs. Parameters are described in the code of the executable using *parameter describers*

  This means that while implementing an executable, for each possible parameter a *parameter describer* has to be defined which describes the parameters names, if they optional or mandatory and which value sets are valid or not.

  Later, when an instance of an executables is created, the parameters to be used have to be bound to a value.
- Slots: Slots are the connection between an executable and it's other artifacts of the cGQM plan. For example, a question provides slots which can bind metrics or other questions. During runtime, the results of the bound slot artifacts can be accessed and it's data used for further computation. Each defined slot has a required role. An artifact to be bound has to be compatible with that role.

  Example: A question executable with the question "What is the average complexity of all project files?" could define a slot named "complexity-Data" with the role "complexity". In the *cGQM plugin describer*, the slot can be bound to a metric which returns complexity measures for all files, for example a "CyclomaticComplexity" metric. During runtime, the question executable can access the complexity measures provided by the metric.

At the actual moment, hackyCGQM supports multiple, predefined slot bindings. This means, that to each slot, multiple artifacts can be bound. In the above example, this could mean that multiple complexity metrics like "CyclomaticComplexity" or "OO-Complexity" could be bound. The implementation of the executable has to define how to handle multiple bindings.

Predefined slot binding in this context means that the bindings have to be defined manually in the plugin describer. This allows easy re-wiring of the cGQM plan. So, for example, if the question in the above example is bound to the cyclomatic complexity, it can easily rewired to the OO-complexity just by changing the the binding definition in the plugin describer.

In contrast to predefined binding, an other strategy for slot binding could be "opportunistic" binding where slots are not bound manually, but are bound automatically by the cGQM framework. As each slot and each artifact has a role attached, the framework can automatically search for all artifacts whose role is compatible with a slot and bind it.

After an executable is actually executed, it returns an result which is of the the type `IResult`.

The class architecture of executables in hackyCGQM is illustrated in Figure 4.1.

4.2.3 Plugin Describer

Plugin Describers are XML files describing the actual cGQM plan contained in a plugin. They define the cGQM artifacts, their relations and parameterizes their executables.

An excerpt of a *plugin describer* is displayed in the following code listing. This excerpt originates from the "buildFailures" example plugin (see 5.1.1). Only two artifacts are contained, the question "Which percentage of NightlyBuilds succeeded in the last 14 days?" and the according metric which delivers the build data.

```
<cgqm id="buildFailures">
    <description>-</description>
```

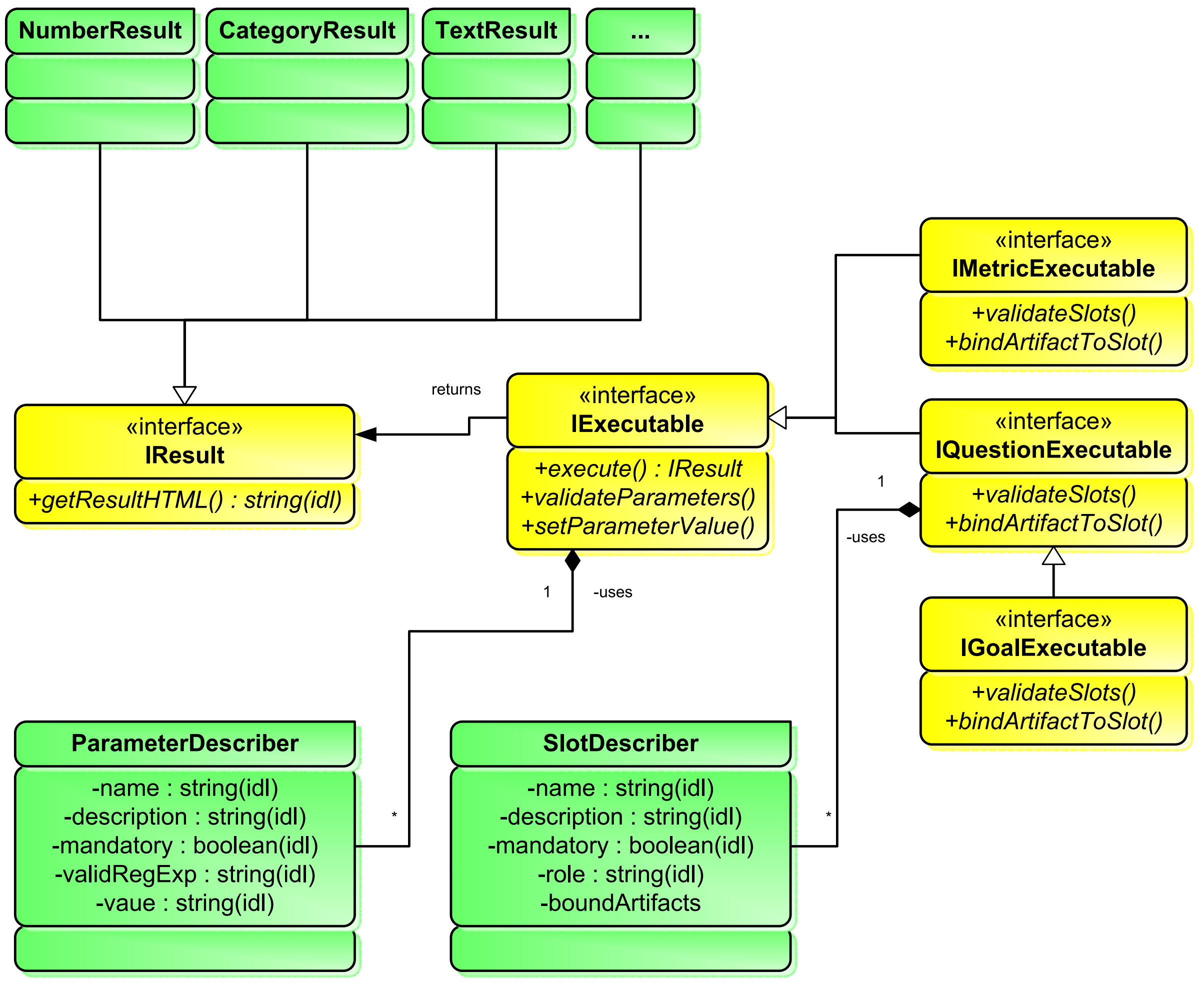


Figure 4.1: Executable interface architecture

The central Interface is `IExecutable`*. Subtypes of this are the interfaces for questions, goals and metrics.* `ParameterDescriber` *and* `SlotDescriber` *are used to customize an executable for a cGQM plan. Executables return subtypes of* `IResult`*.*

```
    <goals>
        <!-- goals omnitted -->
    </goals>
    <questions>
        <question id="q_nightlyBuildFailures">
            <implementingClassName>
                    cGQM.plugin.buildFailures.QNightlyBuildFailures
            </implementingClassName>
            <description>-</description>
            <rationale>-</rationale>
            <roles>
                <role name="nightlyBuildSuccessPercentage"/>
            </roles>
            <parameters>
                <parameter name="answerMode">successPercentage
                </parameter>
                <parameter name="intervalType">Day</parameter>
                <parameter name="startDelta">2week</parameter>
                <parameter name="date">today</parameter>
            </parameters>
            <questionText>
                Which percentage of NightlyBuilds succeeded
in the last two weeks?
            </questionText>
            <slotBindings>
                <slotBinding slotName="buildData"
             objectId="m_buildData"/>
            </slotBindings>
        </question>
    </questions>
    <metrics>
        <metric id="m_buildData">
            <implementingClassName>
                cGQM.plugin.buildFailures.MNightlyBuildData
            </implementingClassName>
```

```
37                 <description>-</description>
38                 <rationale>-</rationale>
39                 <roles>
40                     <role name="buildData"/>
41                 </roles>
42                 <parameters>
43                     <parameter name="userMail">hackystat-l@hawaii.edu
44                     </parameter>
45                 </parameters>
46             </metric>
47         </metrics>
48 </cgqm>
```

Following lines are interesting in this excerpt:

- 8-10: Here, the question is bound to a question executable.
- 13-15: Definition of the questions role (used for binding to the goal)
- 16-22: Setting of the parameters required by the executable
- 23-25: Definition of the question text
- 26-28: These lines bind the metric to the question. The executable has one metric input slot named "buildData" which is bound to the metric "m_buildData" defined in lines 32-46.

A documentation of the XML schema used by the plugin describer can be found in the Appendix Chapter A on page 115.

A *plugin describer* can be written and validated using the the *plugin development environment* described in the next sub-section.

4.2.4 Plugin Development Environment

HackyCGQM provides a basic plugin development environment (PDE). A PDE consists of a folder containing an empty *plugin describer*, build scripts needed to compile, validate and generate the final plugin `.jar` file and all needed java libraries. In addition to that, basic documentation, XML schema files and an Eclipse project definition is included.

The workflow for creating a plugin is as following:

1. Generating a new, empty PDE
2. Importing the new plugin project into Eclipse
3. Coding all executables and the plugin describer
4. Validating the plugin describer
5. Using the provided build scripts to pack the plugin
6. Deploy it to the hackyCGQM server

4.2.5 Plugin Validation

Plugins can be developed independently from the cGQM framework. For supporting and securing the plugin development process, a *validation* mechanism is helpful.

In hackyCGQM, there are two validation mechanisms: The *client validation* and the *server validation*.

The *client validation* is integrated into the *plugin development environment*. It is part of the build process, new plugins will only be created if the client validation passes without failure.

The result of the validation is a validation protocol. It contains all status messages created during validation. These status messages can have one of four severity levels:

- Info: Just an information. Validation continues.
- Warning: Possible failure. Validation continues.
- Error: An error which will prevent the deployment of the plugin. Validation continues.
- Fatal: Fatal error which prevents the plugins deployment and aborts the validation process immediately.

The validation is successful if no errors or fatal errors occurred.

The client validation will primary ensure and check following conditions (only failures and warnings):

- Fatal: Plugin describer's adherence to the XML standard
- Fatal: Plugin describer's adherence to the provided XML Schema Definition (XSD).
- Fatal: Syntactical correctness of the executables
- Error: Formal validity of artifact parameters (correct type, matching regular expression patterns)
- Error: Formal validity of slot binding (role conformance, link conformance)
- Error: Correct type of the bound executables
- Warning: Presence of all mandatory parameters (they could also be provided later, so it's just a warning)
- Warning: Existence of the bound executable (could also be part of another plugin)

A plugin passing the client validation has not to be completely valid as most checks for ensuring validity cannot be performed on client side. Because of that, there is an additional server side validation which will be invoked when plugins are deployed or loaded.

This *server validation* is just an extension of the *client validation* and will additionally check all conditions which are depended on the interaction of plugins (the existence of bound executables or artifacts when they are part of other plugins).

These validation techniques simplify the development and deployment of plugins effectively as many trivial mistake can be detected before the plugins execution.

4.2.6 Plugin Lifecycle

Although the plugin's lifecycle was already indirectly mentioned in the past subsections, it is briefly re-stated here for better clarity:

New plugins can be created by hackyCGQM, including their individual *plugin development environment*. After creation, users can start developing the plugin. This process consists of writing a *plugin describer*, the *executables* and their *documentation*.

After passing the *client validation*, plugins can be deployed to the hackyCGQM server where they are validated again using the *server validation*. After they are successfully deployed, they are stored in the servers plugin repository and can be executed.

This process is also illsutrated in Figure 4.2.

4.3 Data Collection and Metrics

One of the primary aspects of cGQM is the automated collection of required metric data. This means that data has to be collected without manual interaction like filling forms, pressing buttons, using logs or time cards.

For hackyCGQM, Hackystat implements this kind of data collection. When properly configured, Hackystat (see 2.4) unobtrusively collects a wide array of software product and process measures automatically. This is done by using sensor which are either installed on the developers workstations or on central servers, depending on their purpose.

hackyCGQM provides several pre-implemented *metric executables* for accessing the Hacktstat data repository. These executables can easily reused, parametrized or modified for usage in custom plugins.

- telemetry stream access: This executable accesses Hackystat's *telemetry* module. It can deliver high-level data streams, describing selected aspects of the dataset over a longer time scale. It can access arbitrary, valid telemetry streams.
- sensor data access: This executable can access *raw sensor* data inside Hackystat's data repository. It returns a list of sensor data entries of the given timespan of either one specific project member or all project members.

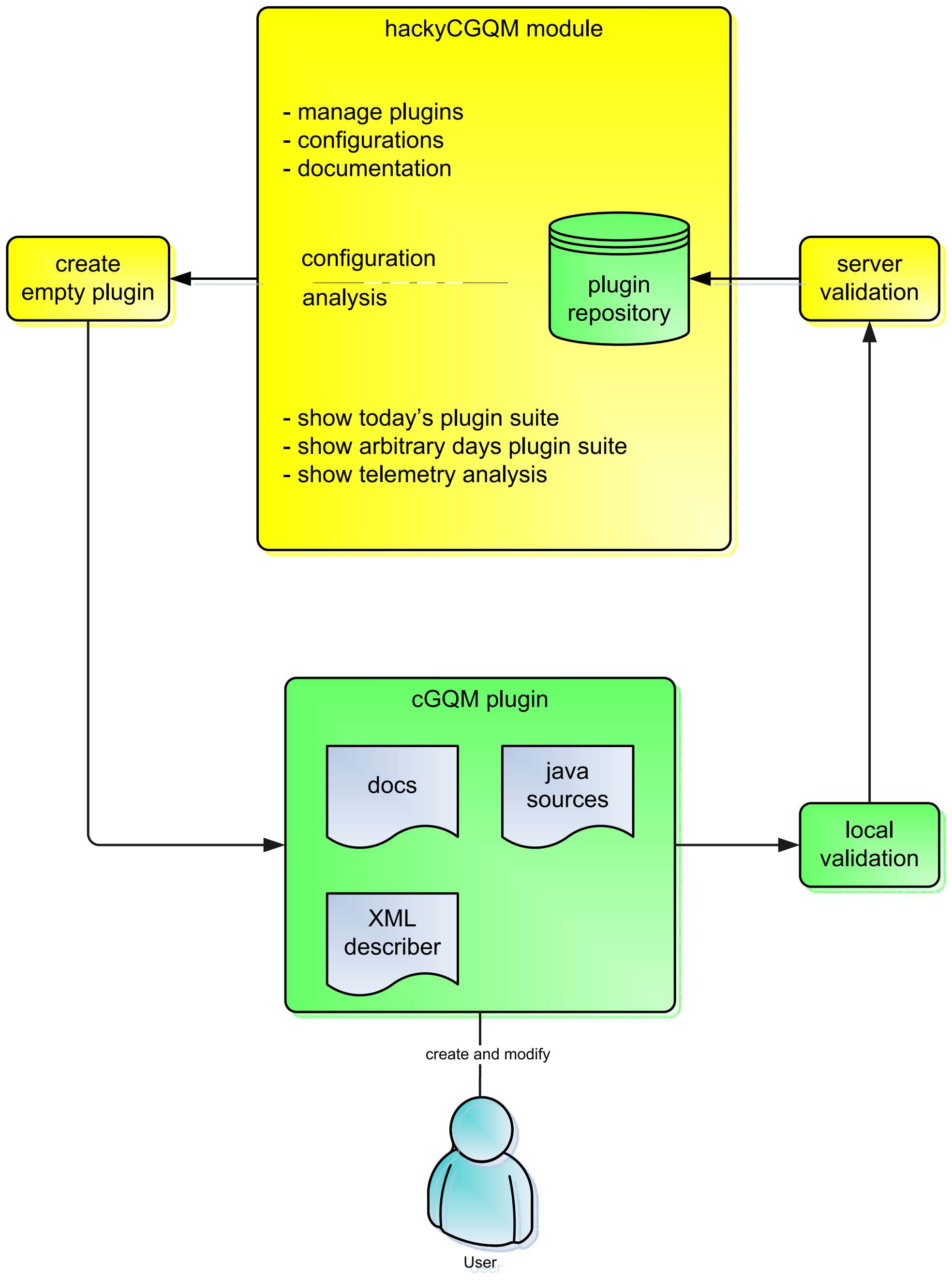


Figure 4.2: Plugin Lifecycle

- daily summary access: *Daily summaries* are summarized statistics of a given sensor data type of a single day. As an example, for the *commit* sensor data type, this could contain "number of commit", "lines added" or "lines deleted" for a whole day. Daily summaries contain the data of one singe project member.
- daily project summary access: This executable returns *daily project summaries*. These are similar to the *daily summaries*, but contain data of all project members instead of just one.

These predefined executables can access the full range of Hackystat measurements and are a suitable starting point for developing plugins.

As a result of technological restrictions and the general usability of the system, the role of metrics in hackyCGQM has slightly changed compared with classic GQM.

In most GQM case studies (as discovered while performing evaluation 5.4), metrics are used in an highly specialized way and are tailored for the single purpose of answering a single question. This is self-evident as GQM's purpose is to create goal-oriented, targeted measurement programs with focusing on metrics which help to answer questions and skipping metrics which don't. Many encountered metrics are like "Defects detected in the system" and "Highly severe defects detected in the system".

As hackyCGQM feeds itself with data automatically collected with sensors, this level of detail becomes impractical. Automatically collected raw data usually contains many aspects. So, for example, the Hackystat issue sensors create a raw measurement entry for each issue in a bug tracking or issue tracking system, containing it's name, date created, severity, user who reported it and other information. If the number of issues is interesting, a question needing this metric could just take all raw sensor data and count the entries. If the number of highly severe issues is needed or those created by a specific user, just the same raw sensor data can be accessed, filtered and processed.

During the development of hackyCGQM, it turned out that these "multidimensional" metrics are much more useful than specialized metrics as the metric implementations can be reused more easily. Also, many different questions can depend just on one metric which speeds up and simplifies the development of plugins.

4.4 Architecture

This section gives a brief overview over hackyCGQM's architecture. The architecture is presented in two parts: a) how hackyCGQM integrates into Hackystat and b) how it works internally.

4.4.1 Hackystat Integration

As already mentioned in chapter 2.4.1, Hackystat implements a micro kernel architecture, as illustrated as a reminder in Figure 4.3 .

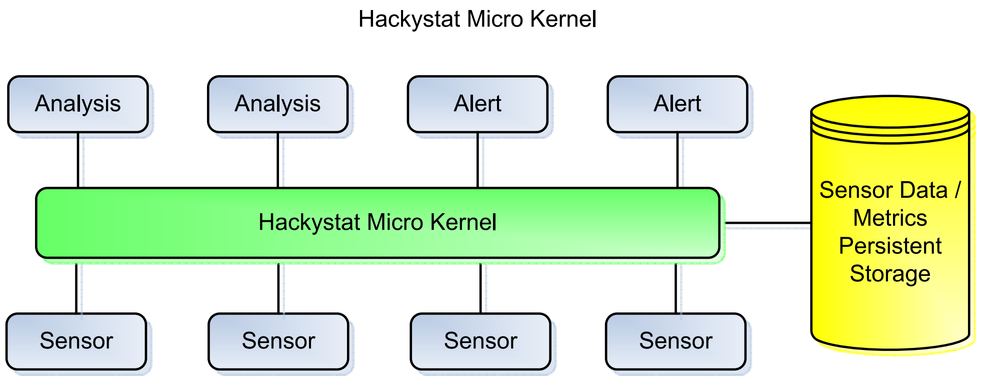


Figure 4.3: The Hackystat microkernel

The micro kernel provides a dynamic extension architecture which allows to plug-in new modules providing new sensors, data types, analysis or alerts.

HackyCGQM is integrated into the Hackystat system as an additional module. It integrates two new analysis pages, one configuration page and a telemetry reducer for Hackystat.

The following list shows the extensions hackyCGQM provides:

- analyis: The *cGQM main analysis* is the primary user interface hackyCGQM provides. It executes a specified cGQM plugin for a specified project and date.

- analyis: The *cGQM plugin documentation* shows the content, structure and documentation provided for a selected plugin.
- configuration: The status of hackyCGQM can be checked using the *cGQM configuration*. There, administrators can access the plugin load logs, the validation logs and can reload all plugins.
- telemetry reducer: As described in 3.7, hackyCGQM provides a telemetry reducer which feeds the telemetry module with data.

The integration of hackyCGQM into Hackystat is roughly illustrated in Figure 4.4.

4.4.2 Internal Architecture

This subsection briefly introduces the internal hackyCGQM architecture.

The hackyCGQM framework (without the sample plugin implementation) consists of 10 top-level packages. These can be grouped into four responsibility groups, as illustrated in Figure 4.5:

- Plugin definition and management
- Executables and results
- Hackystat integration
- Utilities and helpers

In the following course of this subsection, each of these ten top-level packages is briefly described, ordered by their responsibility.

Plugin Definition and Management This package group is responsible for plugin definition and management. It contains two packages: `datamodels.cgqm` and `manager`, both explained below.

- `datamodels.cgqm`: This package contains the data model for representing a cGQM plan. It's main responsibility is to load and represent the *cGQM plugin describer* (see 4.2.3). It primary consists of data classes following the JavaBean standard [38], representing each data aspect describable

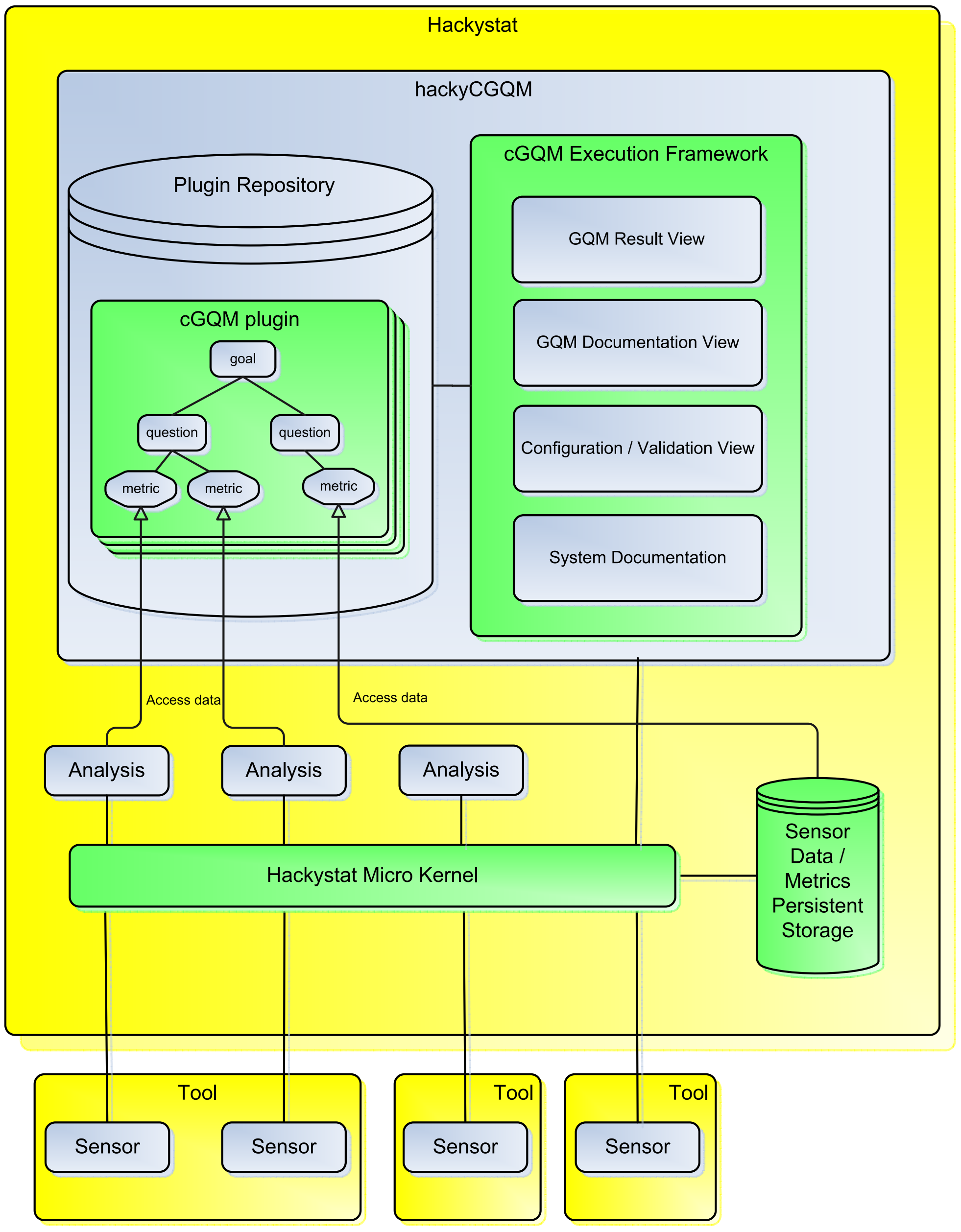


Figure 4.4: hackyCGQM and Hackystat

hackyGQM integrates into Hackystat as a analysis. The metrics of defined cGQM plans can access analysis results (for example telemetry) or raw sensor data.

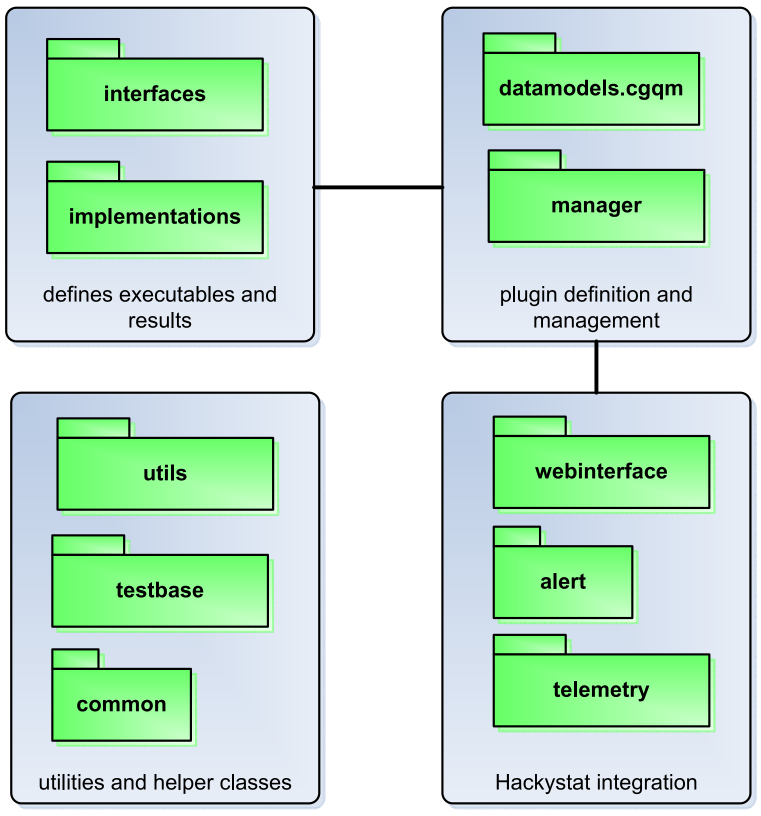


Figure 4.5: The hackyCGQM java top-level packages by their function
hackyCGQM's packages can be categorized to four functional groups.

with the *plugin describer*. The persistence between these data beans and *plugin describer*, which is a XML file, is realized using the JiBX Binding Framework [39].

- `manager`: The `manager` package contain the classes implementing the hackyCGQM plugin repository. This repository is responsible for loading, reloading and validating plugins and makes them accessible to the other hackyCGQM subsystems. One of its classes is also responsible for generating the dynamic overview graphs as illustrated on the screenshot in Figure 5.13.

Executables and Results This package group provides interfaces and abstract base implementations for implementing hackyCGQM *Executables*. Executables are explained in chapter 4.2.2, the layout of the two packages `interfaces` and `implementations` is illustrated in Figure 4.1 (in this Unified Modeling Language (UML) diagram, the `interfaces` classes are yellow and the `implementation` classes are green. The diagram is not complete as it omits the abstract base implementation which can be used to speed up the development process of new *executables*).

Hackystat Integration The packages of this group are responsible for integrating hackyCGQM into Hackystat. It consists of three packages: `webinterface`, `telemetry` and `alert`.

- `webinterface`: The `webinterface` package uses the webinterface hooks provides by Hackystat to integrate the two new analysis (cGQM main analysis, plugin documentation) and the new configuration page (cGQM configuration). It consists of the controller classes and according view-layer Java Server Pages (JSP) pages.
- `telemetry`: This package contains hackyCGQM's telemetry reducer used to feed cGQM data into the telemetry system (see 3.7).
- `alert`: Implements the hackyCGQM alert, used for "push"-style analysis (see 3.5.4).

Utilities and Helpers This last package group contains general purpose packages either used commonly throughout the whole hackyCGQM system or designed to support and simplify the definition of new *executables*. Three packages are part of this group: `utils`, `testbase` and `common`.

- `utils`: `Utils` is the largest of hackyCGQM's packages. It basically provides generic utility classes which are designed to simplify the *executable* development process, but are also used in other parts of the system. It includes for example template engines, a graphic generator, conversion and formatting utilities and utilities for result managing.
- `testbase`: `Testbase` provides abstract base classes for all unit test cases for the whole module. The base test types are standard tests, data-driven tests and server-side tests.
- `common`: This package contains specialized exceptions, extensions for the JiBX binding framework[39] and specialized classloaders for dynamic loading of *executables* in plugins.

This section provides just a brief overview over the hackyCGQM architecture. A complete description would would go far beyond the scope of this document and is not included.

4.5 cGQM and SPCC

HackyCGQM is a system designed for *project control* and *project assessment*. By that, hackyCGQM can be considered as a special case of a SPCC (see 2.3).

A core concept of the G-SPCC reference implementation are *visualization catenas*. These catenas are made up of instances of predefined functions which can be connected to each others to create a data processing network which finally feeds views presenting the data in a role-specific way.

In hackyCGQM, G-SPCC concepts are realized in the following way:

- The concepts of *functions* in hackyCGQM is realized by *executables* (see 4.2.2). In contrast to the G-SPCC reference implementation which uses more generic functions, hackyCGQM uses with the *executables* concept specialized "functions" representing each goal, question are metric.

- Although it is possible in hackyCGQM to connect the *executables* (the hackyCGQM match of the G-SPCC *functions*) to arbitrary shaped networks, they will in most cases turn out to be organized in a 3-layered tree structure. The metrics will be the lowest layer, feeding the second layer with the questions and finally ending with the uppermost layer with the goals.
- Each plugin can be considered as a view as they visualize certain, goal-oriented and role-oriented data (the “role” concept is implicitly realized by defining a “viewpoint” in the GQM goals (see 2.1.1)).

4.6 Statistics

This section just displays some simple statistics for the hackyCGQM project so that one is able to get a rough picture of it’s size and complexity.

metric	**value**	**comment**
active time	185 hours	Active time is not the effort. It is the pure time spend for coding the system.
source lines of code	11707	Lines of code without whitespace and comments
number of classes	198	
number of methods	939	
unit test coverage	80 %	Indirect method level coverage of jUnit tests

Table 4.4: Project end statistics for hackyCGQM

Chapter 5

Evaluation and Demonstration

This chapter demonstrates and evaluates the cGQM reference implementation hackyCGQM. It is separated into two subparts:

- Introduction and demonstration: Introduces several cGQM example measurement programs and their usage.
- Evaluations: In sections 5.2 to 5.4, evaluations focusing on selected aspects of hackyCGQM are performed. These are:
    1. An effort discussion estimating first reference values for the effort needed for performing cGQM measurement programs (Section 5.2)
    2. A mapping evaluation for assessing the potential of cGQM compared to general GQM programs (Section 5.3)
    3. User feedback interviews (Section 5.4)

5.1 Implemented cGQM Examples

At the actual moment, there are three different sample implementations of cGQM programs designed to show different aspects of hackyCGQM and serve different purposes.

These examples are:

1. *buildFailures*: This plugin deals with the problem of failing nightly integration builds. It is used for an effort comparison discussion and an usability evaluation.

2. *issues*: This example answers questions connected to the issue system and it's usage. It is based on a GQM case study conducted by Solingen and Berghout in in 1999 [8].

3. *overview*: This plugin's purpose is mainly to illustrate a non-standard usage of cGQM which analyzes huge amounts of raw sensor data to generate an uncommon overview over a software project.

5.1.1 "BuildFailures" example

Hackystat implements an automated build process for creating the 7 different configurations for Hackystat. A configuration is a specially tailored version of the system with different modules included. These configurations can be understood as a product line. For simplifying the process of maintaining different configurations and insuring their integrity and proper function, the already mentioned automated nightly integration build process was established. This integration build compiles and tests each of the configurations individually each night, as illustrated in Figure 5.1.

Beside creating the configuration's distribution, the main reason for establishing an automated integration build process is defect detection and prevention. During the build, three major phases are performed: The code is checked for conformance with the coding standards using *checkstyle* [40], *compiled* and tested using *junit* [41]. Beside that, some minor, and usually stable, non-failing tasks like javadoc creation or bundling are performed.

A configuration only passes the build successfully if all theses phases pass without error. If an error occurs, the build tries to continue if possible and logs all error messages into a build protocol. As soon as a build does not finish successfully, all developers are notified via email.

This process is designed to find inter-module dependency errors meaning by compiling all modules being part of an configuration and running their unit tests, the integration build is able to find errors which occur in one module, but were caused by a modification of an other module.

This kind of error is not likely to be found by a developer who is modifying the module causing the error as he usually only tests and compiles his module and not all other depended modules (this is due to the expensive testing process, testing all modules can take up to two hours even on a fast machine).

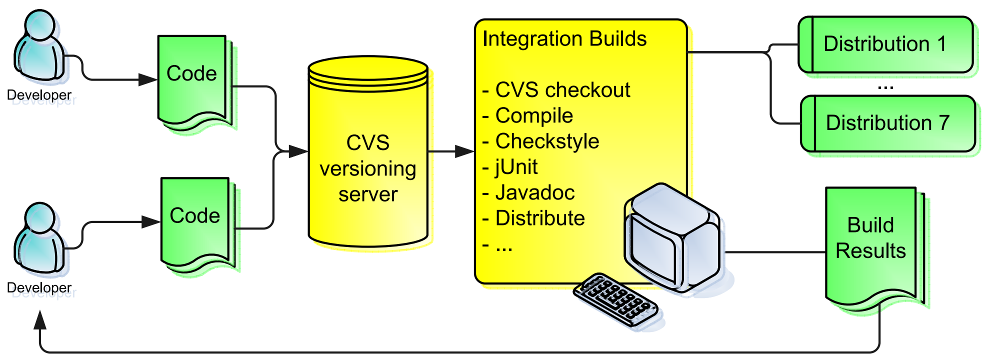


Figure 5.1: The Hackystat build process

Each night, the integration build server creates and tests all seven configuration using the latest code from the CVS repository.

When a nightly build fails, everybody is notified via email. The project manager spends some time for analyzing and locating the failure and notifies the developer who might be responsible for the failure. These developers usually analyze the failure again for themselves and fix them. This process generates a considerable interruption in the development process and is quite expensive as usually several persons are involved. But as long as this process prevents severe failures in the software, it is completely justified.

But one major problem with the integration build process which arose was that the Hackystat integration builds failed too frequently. More than 1/3 of all nightly builds failed in 2004, and in most cases, no real, severe software flaw causing the build failure could be located.
As a consequence of that, considerable effort was wasted just for fixing the build failures.

For solving and understanding the problem, several hypotheses were stated. One assumed that the majority of these build failures was just caused by careless developers who commit flawed code to the CVS. These failures would be completely avoidable if the developer had been more disciplined. This leads to the next assumption that the majority of failures were caused just by a small group of developers with a badly controlled, personal development process.

The "buildFailure" example plugin is designed to clarify the problem of failing builds and finding it's solution. After the rework done during and after the user interviews (see 5.4), it consists of one goal, nine questions and six metrics.

Following artifacts are part of the plugin with dependencies as illustrated in Figure 5.2:

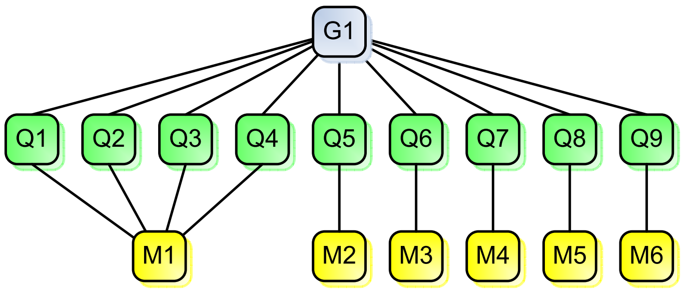


Figure 5.2: GQM structure of the "buildFailures" example

Goal 1: Improve nightly build success rate from the view point of the development team for the Hackystat project at the Collaborative Software Engineering Institute, University of Hawai'i (CSDL).

Question 1: Which percentage of nightly builds succeeded in the last 14 days?
Rationale: The average successrate indicates the actual stability of the build process, important for assessing the actual state of the process.

Question 2: What was the nightly build success rate in 2004?
Rationale: This value serves as a baseline for estimating the improvement compared to the last year.

Question 3: What is the distribution of build failure types in the last 14 days?
Rationale: Knowing the types of occurring build failures can be used for developing strategies for preventing them. For example, if a majority of build failures is caused by trivial problems like

"checkstyle", this indicates the need for enforcing better development processes.

Question 4: Which modules did cause how many build failures of which type in the last 14 days?
Rationale: This question's answer can indicate if there is a problem with a specific module and thus indirectly with a specific developer or group of developers.

Question 5: How many files were committed to the individual modules in the last 14 days? (omitted in effort evaluation)
Rationale: One hypothesis is that the number of build failures is depended on the activity level. Here, number of commits serves as a proxy for activity.

Question 6: How much active time was spend developing the individual modules? (omitted in effort evaluation)
Rationale: As with the last question, activity is approximated here with active time.

Question 7: How many hours did the developers spend on local builds in the last 14 days in average per active person? (omitted in effort evaluation)
Rationale: An other stated hypothesis is that frequent local builds can prevent integration build failures, but they also can waste time. This question shows an overview over how much time developers spend building locally.

Question 8: Which tasks did the developers run how often during their local builds? (omitted in effort evaluation, added during interviews)
Rationale: Local builds only prevent common failure causes like checkstyle or junit when these tasks are performed during the execution. This question visualize who is invoking which build tasks.

Question 9: Which unit tests failed most in the last 2 weeks? (omitted in effort evaluation, added during interviews)
Rationale: This question is based on a vague claim that unit tests which are failing often in local environment also fail the

integration builds and indicate a problem with a module.

Metric 1: Nightly build data (raw sensor data; includes date, time, type, failure message)

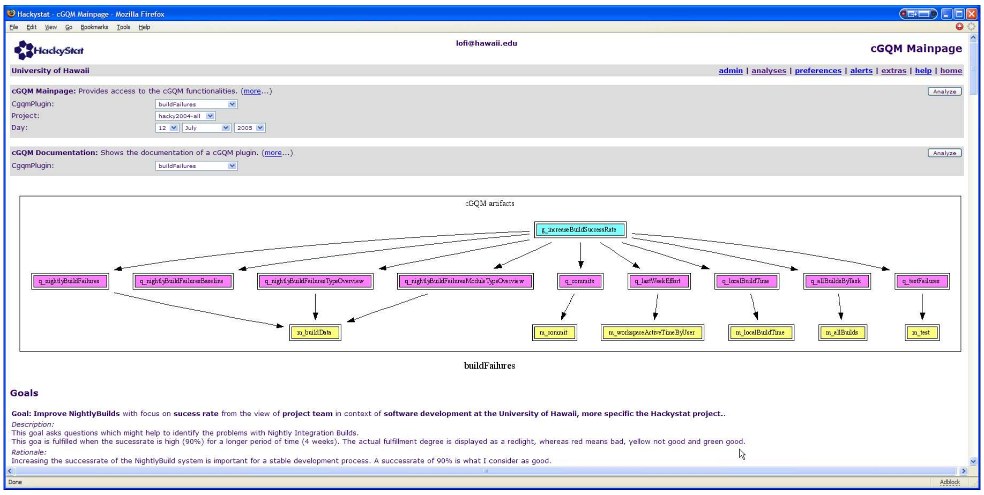
Metric 2: All build data (raw sensor data; includes date, time, type, failure message)

Metric 3: Local Build Time (telemetry stream; summarized local build time for each developer)

Metric 4: Test data (raw sensor data; includes date, time, file, duration, result)

Metric 5: Commit data (raw sensor data; includes date, time, user, file, size)

Metric 6: Workspace active time (telemetry stream; summarized active time for each workspace separated by developer)


Hackystat - cGQM Mainpage - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
lofi@hawaii.edu
cGQM Mainpage
University of Hawaii
admin | analyses | preferences | alerts | extras | help | home
Analyze
Analyze
cGQM artifacts
g_increaseBuildSuccessRate
q_nightlyBuildFailures
q_nightlyBuildFailuresBaseline
q_nightlyBuildFailuresTypeOverview
q_nightlyBuildFailuresModuleTypeOverview
q_commits
q_lastWeekEffort
q_localBuildTime
q_allBuildsByTask
q_testFailures
m_buildData
m_commit
m_workspaceActiveTimeByUser
m_localBuildTime
m_allBuilds
m_test
buildFailures
Done
Adblock

cGQM Mainpage: Provides access to the cGQM functionalities. (more...)
CgqmPlugin: buildFailures
Project: hacky2004-all
Day: 12 July 2005

cGQM Documentation: Shows the documentation of a cGQM plugin. (more...)
CgqmPlugin: buildFailures

Goals

Goal: Improve NightlyBuilds with focus on **sucess rate** from the view of **project team** in context of **software development at the University of Hawaii, more specific the Hackystat project.**.
Description:
This goal asks questions which might help to identify the problems with Nightly Integration Builds.
This goa is fulfilled when the sucessrate is high (90%) for a longer period of time (4 weeks). The actual fulfillment degree is displayed as a redlight, whereas red means bad, yellow not good and green good.
Rationale:
Increasing the successrate of the NightlyBuild system is important for a stable development process. A successrate of 90% is what I consider as good.

Figure 5.3: The "buildFailures" analysis start screen
The analysis page starts with an overview graph showing the cGQM plan, followed by all goals and questions and their answers.

5.1.2 "Issues" example

The "issue" example is inspired by a GQM case study conducted by Solingen and Berghout in in 1999 [8] (named "Case A" in the literature reference).

The “issue” plugin remodels a few of the questions and goals used in the Berghout case study to show cGQM’s ability to perform real world GQM plans. But as there was no real need for this measurement program at the CSDL, the study was not fully reimplemented.

Still, this cGQM plugin shows a quick and clear overview over the usage of Hackystat’s issue tracking system without the unnecessary additional detail originally provided by the Berghout study.

It contains of 1 goal, 4 questions and 3 metrics, connected to each other as illustrated in Figure 5.4. As an example, a chart generated for answering question 2 is shown in Figure 5.5.

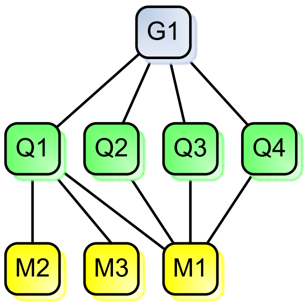


Figure 5.4: GQM structure of the “issues” example

Goal 1: Analyze CSDL Hackystats’s issue tracker system content with focus on the issue handling from viewpoint of the development team.

Question 1: What is the distribution of issues by their severity over modules? What is the complexity/ size of these modules?
Rationale: This question’s answer provides the ability to check for correlations between size/ complexity and number of issues on a module level.

Question 2: What is the distribution of issue types over modules?
Rationale: Helps to analyze which kind of issues are accumulated for a given module. High issue number could either mean that the module is error-prone or that people have many ideas how to improve it or how to implement new features. This question helps to decide which of this is the case.

Question 3: What is the distribution of severity of unsolved issues?
Rationale: Gives an overview over the issues which are to solve in the next time.

Question 4: When were issues reported and what is their actual status?
Rationale: This question's answer shows how quickly issues are solved. In the best case, there should not be any very old issues in the system.

Metric 1: Issue data (raw sensor data; contains date, status, reporter, severity, etc)

Metric 2: Dependency data (used as proxy for complexity; raw sensor data; contains file, ingoing dependencies, outgoing dependencies)

Metric 3: Filemetric data (raw sensor data; contains file, size)

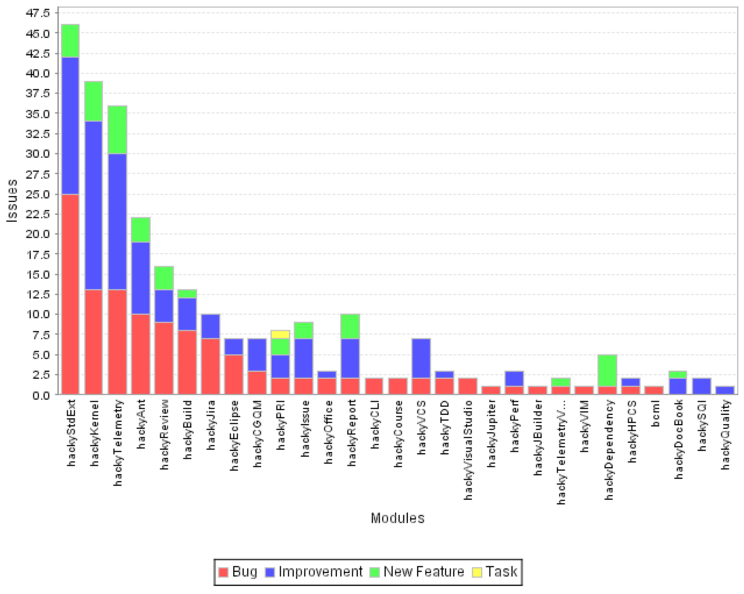


Figure 5.5: A chart generated for the question "What is the distribution of issue types over modules?".

Each bar represents a module, the colors represents the type of the issues.

5.1.3 “Overview” example

The “overview” example is used as a technological “playground”. It’s purpose is to develop a technology and representation for answering the questions “What was going on in my project in the past timespan X? Who worked how much on which parts of the system? How does the actual systems’ code look like?”.

The resulting representation, the *software topography chart*, is still immature and unfinished, but shows potential. The chart visualizes each java package of the system as a rectangle and notes down the times and developers who worked on it. Inside modules, dependencies between packages are indicated by lines (originally, there should also be dependency lines between modules, but this rendered the chart completely unreadable).

In addition to that, each developer is represented by an unique color. Each active package (a package which was modified in the observed timespan) receives an underlayed background octagon with the developers color who spent the most time modifying the package.

One of the primary strength of the *topology chart* is the ability to aggregate and visualize nearly arbitrary metrics like active time, lines of code or coverage.

As a result, a huge graph is generated which shows which parts of the system were active, how the developers effort was distributed and the “shape” of the the single modules. As this graph is very big, it was displayed for a few weeks on CSDL’s telemetry wall, a big composite screen build out of nine 17” LCD displays (see Figure 5.6 and Figure 5.7) There, users can zoom and pan through the graph and examine it at different detail levels.

Although the graph has no real, constant usage yet, it quickly became a topic of many discussions, resulting in various interesting ideas like interactive presentations, 3D models or hierarchy maps.

5.2 Effort Discussion

This evaluation implements and executes an example measurement program (the “buildFailures” example introduced in 5.1.1) with hackyCGQM to get first

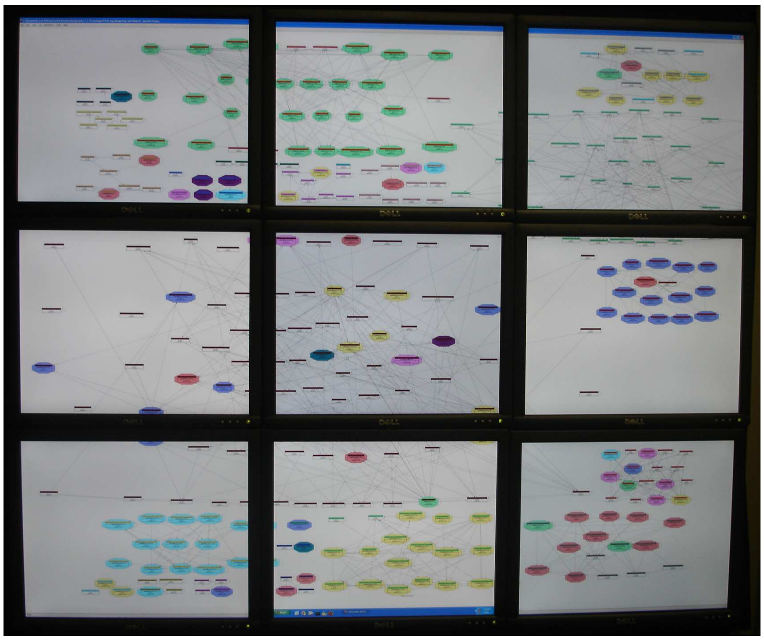


Figure 5.6: The software topography map on the telemetry wall

The topography map spread over 9 monitors on the CSDL telemetry wall.

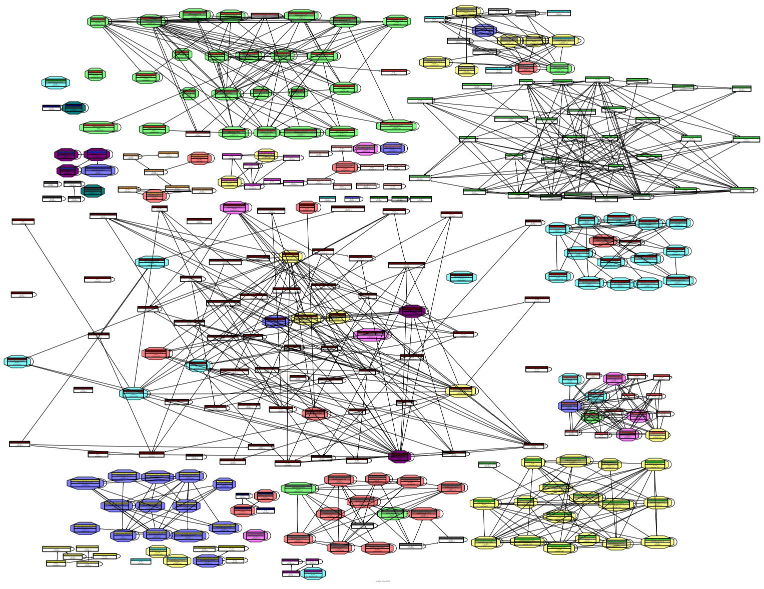


Figure 5.7: The software topography map

This graph is very big - the interactive version can be zoomed so that you can read the four lines of data in each little rectangle reference estimations for the effort to be expected when performing actual measurement programs using the cGQM implementation.

As a comparison, the same program is also performed using a manual approach.

5.2.1 Design

The GQM plan used to perform this comparison is the already introduced "buildFailures" example (see 5.1.1). The person performing it is myself. With that, this evaluation falls into the category of one-shot single subject experiments.

Each execution of a measurement plan consists of two phases, the setup phase where the required tools are prepared and the iteration phase where metric data is collected and analyzed. The iteration phase is repeated every time results are to be presented. In this example, it is assumed that the build failure analysis results are interesting enough to be presented once a week. So, after the initial setup, the execution of one iteration per week is assumed.

For this discussion, it is tried to estimate the behavior of costs if the measurement program is executed over a longer period of time. Each iteration collects, analyzes and presents the data of one week (7 days).

This experiment excludes some questions of the "buildFailures" example for the the manually performed reference measurement ("How much time did the developers spend building locally?" and "Which unit tests failed most in the last n days?"). The reason for this is that the manual collection of data for these questions is too complicated (in the local setting at the CSDL, for answering these questions, approximately 750 log files have to be analyzed per week) and the value of the answers to these questions is not high enough for justifying this effort.
Also, the questions added during and after the user interviews (5.4) are omitted as this evaluation was performed earlier.

In the following paragraphs, the setup for both tests (using hackyCGQM and using a manual approach) of this experiment are described:

cGQM The test of the cGQM plan execution is performed using hackyCGQM. This includes creating a new cGQM plugin. The plugin consists of the *plugin* *describer* and a set of *executables* which perform the data collection and analysis.

Following steps had to be performed:

definition phase:

- Obtaining an empty cGQM plugin: The cGQM system is able to generate an empty plugin project which can be used as a fill in skeleton. This greatly reduces development times as the plugin project automatically provides build scripts, dependent libraries, javadocs of useful helper classes and a validation tool which tests the *plugin describer* for it's consistency. The generated emty plugin can directly be imported to the Eclipse IDE.
- Writing the *plugin describer* XML: The plugin describer is a structured XML file. A XML schema file is provided to help editing the describer and avoiding syntactical errors.
- Implementing the needed goal, question and metric executables: This part of the setup phase consumes the most effort. For speeding up the implementation of executable, hackyCGQM provides abstract base implementations and reference implementations which can be used as a starting point for own executables.
  In the case of the "buildFailures" plugin, eight executables, one helper class and seven unit tests were created.
- Deploy the plugin to Hackystat: While deploying the plugin, it is checked for static errors like wrongly bound metrics or questions, syntactical errors or unimplemented executables.

measurement and analyzing phase:

- executing the plugin: The iteration phase for the cGQM example is very simple. After deploying the plugin, it generates new results everyday completely automatic with no additional effort needed. If necessary, the results can be checked everyday instead of once a week as proposed by the experiment. But it turned out that most people working at CSDL did not consider the daily change in data interesting enough to be checked everyday.

Manual Execution The manual execution of the "buildFailures" example also consists of the two phases *definition* and *measurement and analysis*. The steps needed to perform the test are different from those needed to perform the automated example:

definition phase:

- Write down the GQM plan: The GQM plan is simply written down using Microsoft Word. This contains a short (several lines) descriptions of each goal, question and metric.
- Setup an Excel spreadsheet for data collection and representation: For reducing the effort for data collection, analysis and representation, an Excel sheet is created. The sheet accepts all interesting data in a tabular format and calculates the the required numbers and results automatically as far as possible. Also, charts are drawn automatically. This setup is not really "manual" as most of the computation and visualization is done by the Excel sheet. The only manual interaction needed is filling the data into the columns of the sheet. This tries to simulate a more realistic setting as most people performing GQM won't use pure, manual "pen and paper" techniques.

measurement and analyzing phase:

- Measure the required data: Luckily, it turns out that there is only one data source needed to answer all questions. This data source are the log files which are created by CruiseControl, the system performing the nightly integration builds. For each build configuration, one log file per night is generated. At the actual moment, there are 7 configurations, but in average it turns out that only 3 of them are build per night (only those with changes in them) which makes 21 log files per week in average.
- Enter the data: After accessing the build logs, they have to be analyzed for the required data. After that, the data is simply entered into the Excel spreadsheet. After entering all needed data, the sheet automatically calculates the values needed to answer the question and plots the required charts.

The setup of this evaluation could be considered as "manual GQM friendly", as it does not exploit the strengths of cGQM. The iteration time is assumed as

	cGQM	manual
setup	14h	3h
time per week	0h	1 1/4h
time in 2 month	14h	13h
time in 4 month	14h	23h
time in 6 month	14h	33h

Table 5.1: Time needed for performing the 'buildFailure' example manually and with cGQM. Every week, one analysis just based on that weeks data is assumed.

one week, although cGQM could generate new results every day or every hour without any overhead while manually performed GQM programs would run into problems while trying that. Also, the strength of being able to crunch massive amounts of data (like analyzing 750 log files per week for the excluded questions in this example or processing several hundreds MB of raw data for generating the software topography map used in the "overview" example) is not exploited.

5.2.2 Results

This subsection illustrates the results of the experiment. First measured effort numbers are stated, followed by personal impressions of the execution of the experiment.

Measured times The timed I needed to perform both tests were timed, the results of that timing are shown in table 5.1.

As expected, it turned out that the initial setup costs for cGQM are high compared with manual GQM. But this disadvantage disappears after 9 weeks of usage as manual GQM needs additional effort for each data set which is to be processed.

The measured times for implementing the cGQM example was 14 hours for setting up 1 goal, 9 questions and 5 metrics.

A graph illustrating the extrapolated values of effort needed to execute the analysis depending on the number of weeks is illustrated in Figure 5.8.

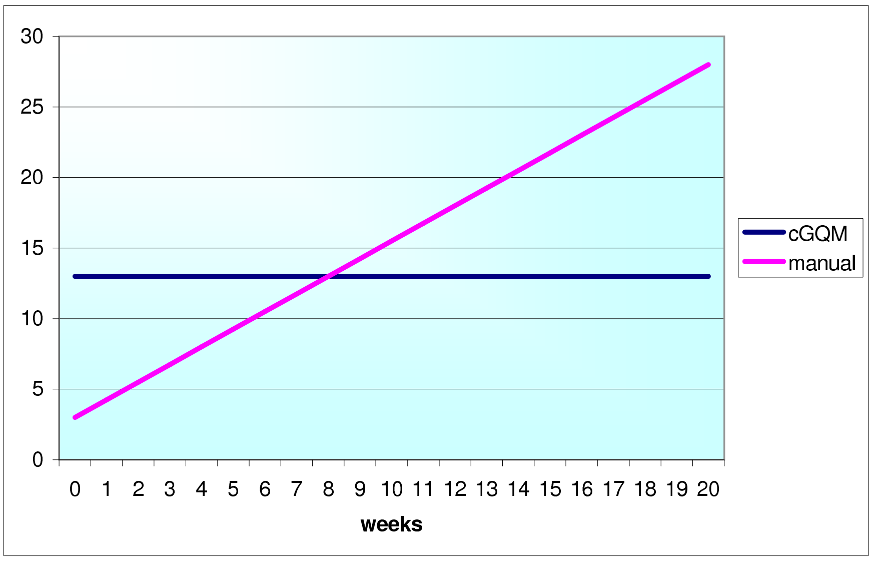


Figure 5.8: Time needed to execute the 'buildFailure' example.
Automated GQM pays of after 2 month in this setting.

Impressions After performing both the cGQM and the manual approach, I tried to write down my own impressions of the execution process of the manually performed measurement program and the cGQM program. This just serves the purpose of providing a better insight into the execution of cGQM based measurement programs.

cGQM

- The setup of the cGQM plugin was a very tedious and annoying work. It took several days to finish and tune the plugin, a lot of refactoring and bug fixing had to be done. The system was not as easy to use as I wished it to be. I am probably the most competent person for implementing a cGQM plugin (as I designed the hackyCGQM framework), but still, the implementation of the plugin was difficult for me.
- After the setup, the plugin performed neatly and didn't need any additional maintenance.

Manual Execution

- The initial setup turned out to be quite simple (setting up the Excel spreadsheets and diagrams)
- The data collection and analysis phase turned out to be extremely annoying. Every morning, I was checking and analyzing all build protocols. This took an average of 3 minutes per protocol with an average of 3 protocols per day.
- The manual data collection turned out to be highly error prone (transcription errors, read errors, categorization errors). The results differed from the ones calculated by cGQM (which are considered as being correct). This also matches the results already found in another study [19] which criticizes the low reliability of manual measurement and analysis results .

I performed the manual GQM execution for one week (one iteration). In my personal opinion, I felt more comfortable with coding the cGQM plugin despite all it's hassles than with analyzing the build logs everyday.

5.2.3 Threats to Validity

The validity of this evaluation is not very high. It is primary designed to just provide a rough effort baseline and give a clue about how fully automated measurement programs relate to manual ones when effort is the only aspect.

This evaluation is specially restricted as it was designed as a one subject, one task evaluation. The main weaknesses are:

- only one subject participated on the evaluation (me as the developer of hackyCGQM). Others might not be as familiar with the system as I am and their times needed to create the cGQM plugin would probably be longer.
- only one measurement program was implemented during the course of this evaluation. The underlying GQM plan greatly influences the measured times. The actually chosen program ("buildFailures" example) was chosen under the consideration that it is a "fair" example for this evaluation. It can be performed automatically so cGQM is usable, but it does not rely on cGQM's special strength of very short feedback cycles or processing huge

amounts of data. But for providing valid results, multiple and different measurement programs have to be evaluated.

5.2.4 Discussion

The implementation of the cGQM example took 14 hours. This number has to be handled with extreme care as other programmers not being also the author of the framework might need much longer and the example itself was not too big (1 goal, 9 questions, 5 metrics). But the needed effort should be reducible by re-using *executables* or *plugin describer* fragments.

This evaluation also confirms an other obvious claim: cGQM can be more efficient and can save effort in settings with many measurement cycles and/or big amounts of raw data which can be measured and analyzed automatically.

In the setting of this evaluation, cGQM pays of after 2 month of usage if effort is the only observed property. But the evaluation's design is somehow biased so that the GQM program with the manual measurement gains advantage.

Further informations about the costs of traditionally performed GQM programs have been collected by A. Fugetta in an industrial setting [42]. There, the same GQM program was executed in sequence in two different projects. The whole GQM plan was designed, developed and executed during the first project. There, it turned out that data collection accounted for nearly 35% of all costs of the GQM program which equaled 2% of the overall effort of all project members.

In the second project, the plan and technologies developed for the first were just reused. The overall effort could be reduced by 40% by reusing the existing plan and environments, but the effort needed for data collection remained the same. So, in the repeated application of the GQM program, the effort for data collection accounted for more than 65% of the overall effort.

In that case study, the CEFRIEL tool (see 2.1.6) was used to support the measurement plan execution. Despite the use of the tool, the measurement program remained costly which is due to the high costs of measurement. While using cGQM, no costs for measurement accrue.

If a GQM program can be performed by cGQM, it can be executed without costs for data collection and analysis, saving a major fraction of the costs usually connected to GQM programs.

5.3 Structural Evaluation

CGQM can only used in cases where measurement and analysis can be performed automatically. This evaluation tries to find out how probable this case is. For doing this, several published GQM-based case-studies are examined and assessed regarding their potential for automation.

5.3.1 Design

This evaluation uses six published GQM studies for performing an assessment if the stated goals, questions or metrics could be mapped into a cGQM plan or not.

These reference case studies are:

Study 1: Solingen and Berghout, Case A [8]

Study 2: Solingen and Berghout, Case B [8]

Study 3: Solingen and Berghout, Case C [8]

Study 4: Solingen and Berghout, Case D [8]

Study 5: Fugetta et al. [42]

Study 6: Lindstroem, Version 2 [43]

Each of the goals, questions or metrics occurring in these case studies are assessed according to

1. if they can be mapped to the actual cGQM implementation (hackyCGQM)
2. if they can be mapped to future, improved cGQM implementations

The degree to which they can be mapped is expressed with three different tokens:

1. **+** : The GQM artifact can be mapped with cGQM

2. **0** : The GQM artifact can be mapped, but restrictions apply (certain aspects can not be mapped, some information is missing)

3. **-** : The GQM artifact cannot be mapped (analysis too complex, metrics depend on data which cannot be collected unobtrusively and automatically)

The main aspect while considering if a given artifact could be mapped to a future cGQM implementation or not is if it can be executed in an unobtrusive and automatic manner. So for example, metrics which relay on a personal opinion of an expert cannot be executed unobtrusively and automatically as you have to ask the expert every time you need a new value for that metric.

Same theoretically applies to questions which cannot be formalized to a sufficient degree. In the case of questions, a question is considered as being mappable (**+**) if all metrics connected to it are mappable and if a visual representation of the metric data can be generated that is sophisticated enough for deriving the questions answer. For example, if the question was: "Is there a correlation between X and Y?", then it is sufficient if it is possible to plot a clear chart showing a correlation or not. This does not exactly answer the question, but it should be enough for this evaluation.
Same also applies to goals.

Example:
One metric found in one of the GQM case studies is "Reviewer's knowledge of the document before the review". This metric can obviously not be measured automatically - not now and not even with better, automated measurement tools. So, this metric is rated a "**-**, **-**".

An other metric found reads like "For each fault detected in the system, classification by severity, module, life-cycle phase and reporter". In the actual version of hackyCGQM, the *Hackystat issue sensor* provides information related to defects like severity or module. But it does not provide life-cycle phase or reporter. So this metric is just rated "**0**" in the "actual state" classification. But, there is no reason why a future implementation could not provide the missing data, so it receives a "**+**" for future versions of cGQM frameworks.

5.3.2 Results

The summarized results of this evaluation can be found in table 5.2. A detailed overview over these results can be found in the appendix chapter B in table B.1.

It turned out that with the actual cGQM implementation 40% of goals, questions and answers can be implemented. With an improved version of cGQM, this percentage increases to 45%.

	+	0	-	+	0	-
Study 1						
Goals	0	1	1	0	1	1
Questions	7	5	10	11	3	8
Metrics	37	3	29	42	1	26
Overall	44	9	40	53	5	35
Study 2						
Goals	1	0	0	1	0	0
Questions	15	4	4	15	4	4
Metrics	17	0	10	17	0	10
Overall	33	4	14	33	4	14
Study 3						
Goals	0	0	1	0	0	1
Questions	0	0	6	0	0	6
Metrics	0	0	16	0	0	16
Overall	0	0	23	0	0	23
Study 4						
Goals	0	0	1	0	0	1
Questions	0	0	2	0	0	2
Metrics	0	0	14	0	0	14
Overall	0	0	17	0	0	17
Study 5						
Goals	2	3	0	2	3	0
Questions	13	0	22	13	0	22
Metrics	0	0	0	0	0	0
Overall	15	3	22	15	3	22

	+	0	-	+	0	-
Study 6						
Goals	2	0	4	2	2	2
Questions	9	3	19	12	4	15
Metrics	28	0	40	34	2	32
Overall	39	3	63	48	8	49
All studies						
Goals	5	4	7	5	6	5
Questions	44	12	63	51	11	57
Metrics	82	3	109	93	3	98
Overall	131	19	179	149	20	160
All studies %						
Goals	31%	25%	44%	31%	38%	31%
Questions	37%	10%	53%	43%	9%	48%
Metrics	42%	2%	56%	48%	2%	51%
Overall	40%	6%	54%	45%	6%	49%

Table 5.2: Results of the structural mapping. Left columns shows results with actual cGQM, right columns results with possible future cGQM implementations.

5.3.3 Threats to Validity

This evaluation gives a first clue about how high the potential for using cGQM might be. Although this evaluation covered many goals, questions and metrics which were extracted from six different case studies, their representativeness is questionable.

Four of the six studies were all conducted by the same persons (Solingen and Berghout) which could bias the results.

But when examining the mapping result distribution between the case studies, it indicates that these studies might cover a larger and thus more representative range of GQM artifacts. This is confirmed by the high variance in the percentage

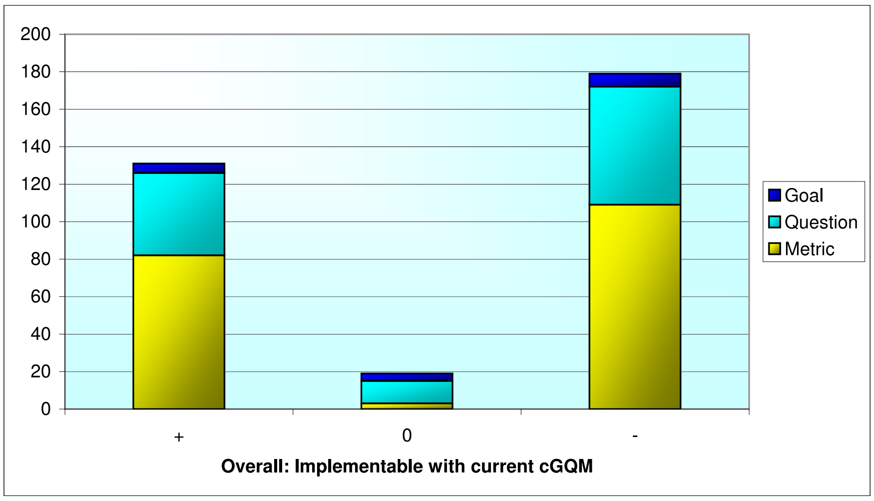


Figure 5.9: Mapping ability summary of current cGQM for examined Goals, Questions and Metrics

+ means a mapping is possible, 0 means a mapping is possible with restrictions, - means a mapping is not possible

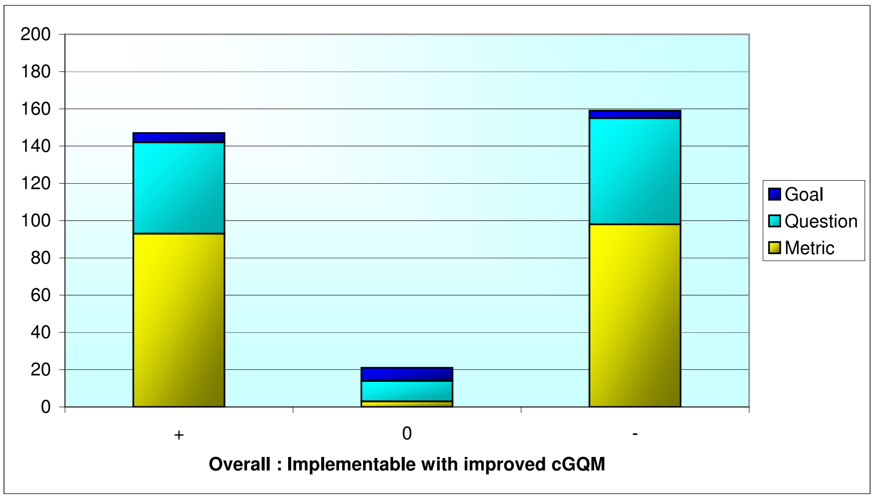


Figure 5.10: Mapping ability summary of future, improved versions of cGQM for examined Goals, Questions and Metrics

+ means a mapping is possible, 0 means a mapping is possible with restrictions, - means a mapping is not possible

of goals, questions and metrics which are implementable with cGQM between the single case studies, ranging from 0% up to 65%.

5.3.4 Discussion

One number in the preceding table is of major importance for the result of this evaluation: Only 45% of all GQM artifacts (goals, questions and metrics) encountered in these six case studies could probably be implemented with cGQM. This clearly shows that cGQM is not a replacement for traditional GQM approaches and implementations, but a specialized paradigm which performs well only on a subsection of the problems where GQM is usable.

But still, this subsection of cases where cGQM is usable is approximately 45% as big as the set of all cases. This shows that automated measurement programs have great potential desipte their inability to replace manual programs.

This limitation originates from cGQM concentration on automatic measurable and interpretable metrics and questions. Already van Solingen stated in [8]:

> "The most valuable information usually comes from people, not from tools."

But this does not mean that cGQM cannot provide valuable information. cGQM shows great strength with it's alternative process models (3.5) and in cases with huge amounts of data. Another aspect to keep in mind is that Hackystat which provides the automated measurements, is ways more sophisticated and powerful than anything mentioned by, for example, van Solingen. Although it is not possible to proof, I strongly belief that it might be the most powerful automated software metric collection framework in existence as it uses other, existing data collection tools and aggregates them into one framework. In addition to that, it is actively under development and new data sources are constantly added. But even the most sophisticated automated data collection tool can probably not provide the same information a human expert could.

The solution for this situation could be the integration of manual data when needed. This contradicts the cGQM Paradigm as it was supposed to shorten feedback cycles and cut costs by introducing complete automation. The effect of integrating manual measurement and/or analysis into cGQM is a topic requiring additional research.

5.4 User Interviews

This evaluation introduces hackyCGQM to potential user groups. It is implemented as a guided 1-on-1 walk-through interview. The purpose of this evaluation is to assess user reactions to several aspects of the hackyCGQM implementation such as usability, understandability or usefulness. A special focus is on the users reactions to presented data and their opinions how to improve the presentation in specific and the overall implementation in general.

5.4.1 Design

In this evaluation, several individuals representing potential user groups are interviewed. During these interviews, the interviewer (me) guides the users through a selected usage scenario. The subject is supposed to "think aloud", sharing his feeling, suggestions and insights with the interviewer. Interesting, upcoming aspects are discussed. All interviewed subjects are performing similar tasks and certain, predefined questions are stated to each of them, but depending on the subjects reactions and upcoming discussions, each interview can focus on different aspects.
Each interview consumes slightly more than an hour in average. The interviews are recorded for later analysis.

Following subjects participated in this evaluation:

1. The Hackystat project manager (representing a user highly experienced with software development, project management, measurement programs and data interpretation. Also, he is experienced and familiar with using Hackystat and interpreting the data collected with Hackystat. He has extensive knowledge of the development process used in the Hackystat project.)

2. Three Hackystat lead developers (representing experienced software developers who are used to work and interpret software metrics on a regular

basis. They are familiar with using Hackystat and have also a extensive understanding of Hackystat's developement process.)

3. Two undergrad students who will join the Hackystat development team (average skilled, less experienced developers with no significant knowledge of neither using the Hackystat system nor developing it)
4. One computer science graduate student (representing an average skilled developer with general software knowledge, but with no special experience in software engineering or Hackystat)

The selected scenario of this evaluation is the already introduced "buildFailure" example (see 5.1.1). The selection of this example increases the significance of this evaluation as this example deals with a problem which is of personal importance to all Hackystat project members (4 of 8 of all participants).

The walk-through scenario simulates a *retrosepctive* (see cGQM process models, 3.5.3) analysis of the Hackystat development process. The analysis focuses on the nightly integration builds and the time span between March and Juli 2005.

The interviews are performed as following:

1. Depending on the subject's experience and familiarity with the system and the development process being examined, the interviews start with a brief to thorough introduction describing the context of the walk-through performed in the interview.
2. Using a telemetry stream analysis of the plugin's goal and and the question "What is the build success percentage of the last 14 days", the subject picks five dates between March 2005 and Juli 2005 which seem interesting to him (see Figure 5.11).
3. Each of these five dates are examined by the subject using the cGQM "buildFailures" analysis with a 2 week time frame (this means, that all questions in the analysis only use data of the two prior weeks before the picked date). This simulates the usage scenario of having frequent measurement cycles which are focused on short term data. Screenshots of this can be found in Figure 5.12, Figure 5.13, Figure 5.14 and Figure 5.15.
4. The whole time span between March and Juli is examined using the same

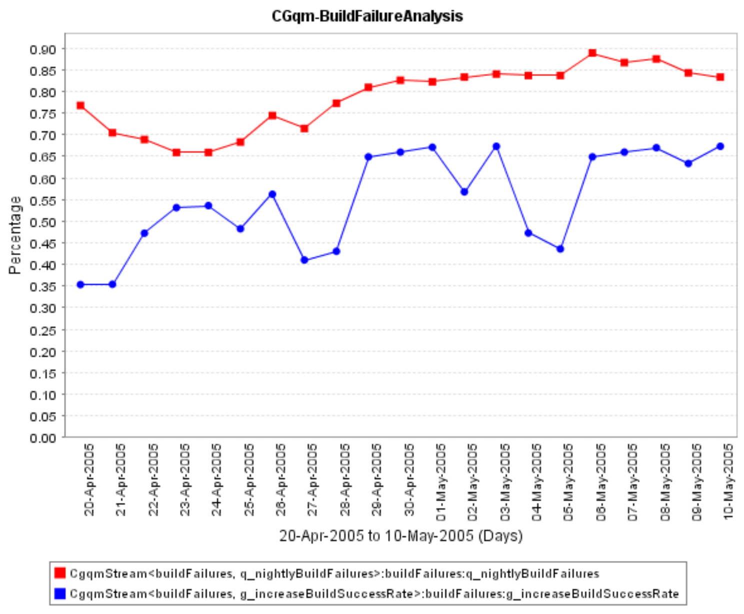


Figure 5.11: hackyCGQM screenshot: telemetry
Screenshot of a telemetry chart [36] used for continous analysis

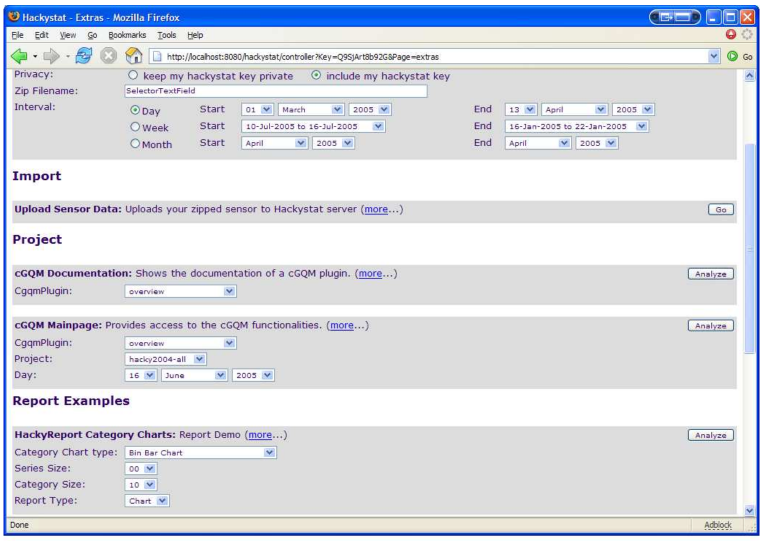
cGQM "buildFaiures" analysis as in the last step, but with a five month time frame (all questions use five month of data and summarize it).

5. Questions and Discussions

The interviews are freely guided by a questionnaire which be found in appendix chapter C. This questionnaire is just a guideline, the questions contained in it are freely integrated into the interview as they fit.

The questionnaire's design tries to eliminate the bias resulting from personal acquaintance between the interviewer an the subjects. This is done by omitting direct qualitative or quantitative questions (like "Do you think this technology is useful? Rate from 1 to 5"). Instead, the questionnaire uses mainly "enumerative" questions focusing on negative or positive aspects of hackyCGQM, the usage scenario or the cGQM method itself (like "Which aspects of the representation of the results did you NOT like?").

By doing this, the questionnaire hopefully produces useful feedback for exploring the weaknesses and improvement areas of cGQM and hackyCGQM, but also highlights strengths and benefits from a users point of view.


Hackystat - Extras - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/hackystat/controller?Key=Q9SjArt8b92G&Page=extras
Go
Adblock
Done

Privacy: keep my hackystat key private [✓ include my hackystat key]

Zip Filename: SelectorTextField

Interval: [✓ Day] Start 01 March 2005 End 13 April 2005

Week Start 10-Jul-2005 to 16-Jul-2005 End 16-Jan-2005 to 22-Jan-2005

Month Start April 2005 End April 2005

Import

Upload Sensor Data: Uploads your zipped sensor to Hackystat server (more...) Go

Project

cGQM Documentation: Shows the documentation of a cGQM plugin. (more...) Analyze

CgqmPlugin: overview

cGQM Mainpage: Provides access to the cGQM functionalities. (more...) Analyze

CgqmPlugin: overview

Project: hacky2004-all

Day: 16 June 2005

Report Examples

HackyReport Category Charts: Report Demo (more...) Analyze

Category Chart type: Bin Bar Chart

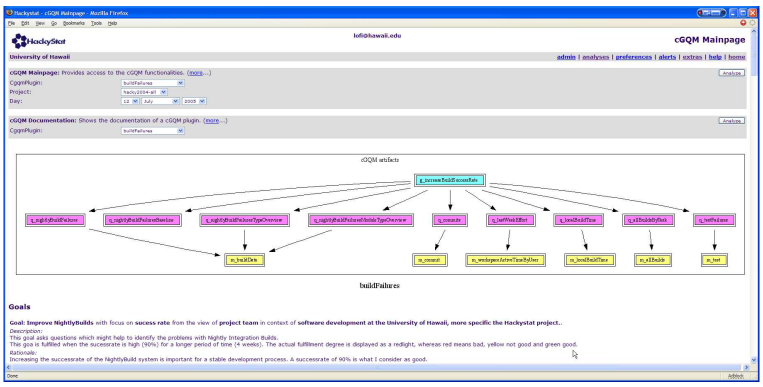
Series Size: 00

Category Size: 10

Report Type: Chart

Figure 5.12: hackyCGQM screenshot: analyis page

Hackystat extra analysis page, start point of cGQM analyses


Hackystat - cGQM Mainpage - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
cGQM artifacts
g_increaseBuildSuccessRate
q_nightlyBuildFailures
q_nightlyBuildFailuresBaseline
q_nightlyBuildFailuresTypeOverview
q_nightlyBuildFailuresModuleTypeOverview
q_commits
q_lastWeekEffort
q_localBuildTime
q_allBuildsByTask
q_testFailures
m_buildData
m_commit
m_workspaceActiveTimeByUser
m_localBuildTime
m_allBuilds
m_test
buildFailures
Done
Adblock

lofi@hawaii.edu

cGQM Mainpage

University of Hawaii

admin | analyses | preferences | alerts | extras | help | home

cGQM Mainpage: Provides access to the cGQM functionalities. (more...) Analyze

CgqmPlugin: buildFailures

Project: hacky2004-all

Day: 12 July 2005

cGQM Documentation: Shows the documentation of a cGQM plugin. (more...) Analyze

CgqmPlugin: buildFailures

Goals

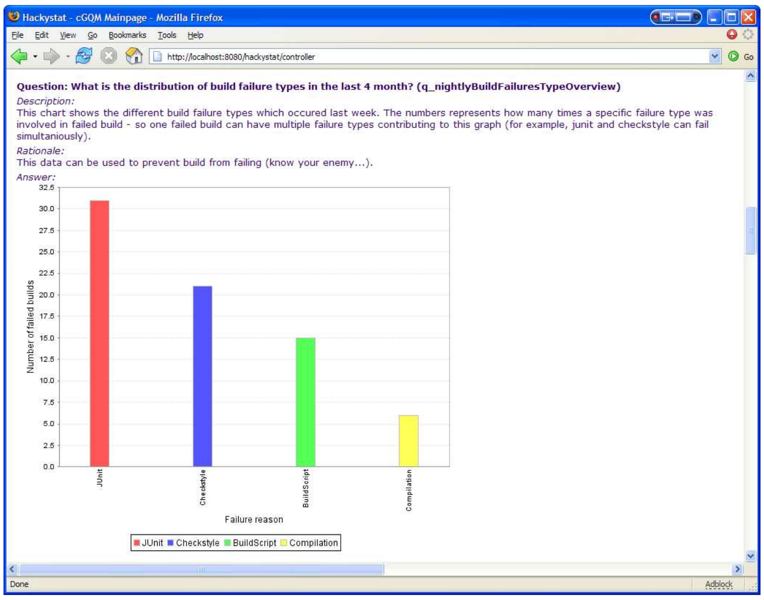
Goal: Improve NightlyBuilds with focus on **sucess rate** from the view of **project team** in context of **software development at the University of Hawaii, more specific the Hackystat project.**.

Description:
This goal asks questions which might help to identify the problems with Nightly Integration Builds.
This goa is fulfilled when the sucessrate is high (90%) for a longer period of time (4 weeks). The actual fulfillment degree is displayed as a redlight, whereas red means bad, yellow not good and green good.

Rationale:
Increasing the successrate of the NightlyBuild system is important for a stable development process. A successrate of 90% is what I consider as good.

Figure 5.13: hackyCGQM screenshot: goal

Screenshot of the 'goal' section of the cGQM example


Hackystat - cGQM Mainpage - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/hackystat/controller
Go
Number of failed builds
32.5
30.0
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
JUnit
Checkstyle
BuildScript
Compilation
Failure reason
🟥 JUnit 🟦 Checkstyle 🟩 BuildScript 🟨 Compilation
Done
Adblock

Question: What is the distribution of build failure types in the last 4 month? (q_nightlyBuildFailuresTypeOverview)

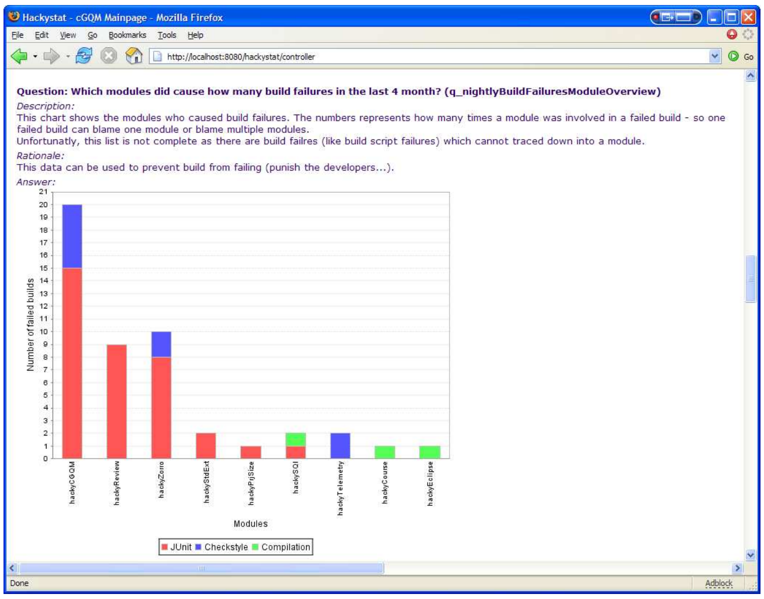
Description:
This chart shows the different build failure types which occured last week. The numbers represents how many times a specific failure type was involved in failed build - so one failed build can have multiple failure types contributing to this graph (for example, junit and checkstyle can fail simultaniously).

Rationale:
This data can be used to prevent build from failing (know your enemy...).

Answer:

Figure 5.14: hackyCGQM screenshot: build failure types

Answer of question 3: "Which type of build failures did occur ?"



Question: Which modules did cause how many build failures in the last 4 month? (q_nightlyBuildFailuresModuleOverview)

Description:
This chart shows the modules who caused build failures. The numbers represents how many times a module was involved in a failed build - so one failed build can blame one module or blame multiple modules.
Unfortunatly, this list is not complete as there are build failres (like build script failures) which cannot traced down into a module.

Rationale:
This data can be used to prevent build from failing (punish the developers...).

Answer:

Figure 5.15: hackyCGQM screenshot: build failure modules

Answer of question 4: "Which modules did cause which build failures?"

If an interview reveals a flaw in the system which is easy to be fixed, these fixes are applied for before the next interview.

5.4.2 Results

This subsection presents the results of the user interviews. They are divided by the four feedback areas which were part of the interview.

The first area focuses on the presented sample cGQM plan and it's presentation. The second area is about general usability of the hackyCGQM framework. For the third evaluated area, questions about future implementations or plugins are asked. Finally, area four contains all aspects which were not part of the previous three.

The "buildFailure" example The "buildFailure" example plugin is already explained in chapter 5.1.1. As the usage of that plugin was the central part of the interview walk-through, most of the collected feedback is related to the plugins design, it's results and it's presentation.

All subjects which were part of the Hackystat development team were highly interested in the results of the cGQM analysis. Before this evaluation, the only known fact was that there are too many integration build failures. Beside that, a wide variety of assumptions, myths and tales about the problems nature existed, but none of them were backed by real data. As the failing builds affect and disrupts each developers workflow, the Hackystat team members were motivated and able to interpret, discuss and improve the presented data.

In contrast to that, the subjects not being part of the Hackystat team were slightly interested, but stayed more passive as analyzing the problem did not affect them directly. As a conclusion of that, the following results in this subsection only refer to members of the Hackystat team.

All for project member subjects rated the analysis as very useful. All of them were surprised by the presented data as it showed new, yet unknown aspects and disproved some common assumptions.

Some quotes recorded from subjects, after being confronted with the build failure reason analysis, were:

> "Wow, if I knew this before, I would have been more careful."

or

> "Oh, I definitely should change my development style and should start doing xxx and yyy ..."

Every subject was able to understand the presented data, interpret it and draw conclusions. During the interviews, the subjects developed four new hypotheses and five new questions, three of them were implemented during the interview. The new, proposed questions were added to the cGQM plan between the interviews and the next subjects and the one who proposed the question (while reviewing the implementation) confirmed the added value.

All subjects mentioned the goal-oriented approach as a positive aspect. They liked the aggregation of data from different sources and their representation as questions and answers. Also, the provided question descriptions and rationales were considered as being valuable.
The major problem with the presented questions mentioned by nearly all subjects was that although the question answers provide a goal-oriented high-level abstraction of the measured data, a need for manual interpretation still remains. Most subjects stated that the questions are very helpful, but they are not and probably can never be the "golden bullet" or "magic sphere".

After being confronted with the question whether the long-term analysis covering five month or the five short-term analysis just covering two weeks each were more useful to them, all subjects were undecided. After asking them which one of the two they would omit if they had to omit one, 3 in 4 subjects answered that they don't want to omit one of the analysis as they like both and they complement each other. The 4th subject finally decided to omit the long term analysis.

All subjects felt that having a short-term analysis which can be executed frequently on arbitrary dates is very useful, although it might not provide enough data without a summarizing long-term analysis.

Two of the subjects directly mentioned that they loved the "explorative" style of analysis where they could freely decide which two week period of the last 5 month they want to see based on the telemetry overview chart (this refers to the *retrospective process model*, see 3.5.3). The other two subjects mentioned this after asking them if they liked the short- or the long-term analysis better.

The *goal fulfillment degree* (see 3.3) calculated by the plugin's goal also became a focus point of the subjects' interest. In the original version of the plugin, the fulfillment degree was presented as a percentage number together with a chart describing how this percentage was calculated. This confused the first subjects as this "arcane" number did not have an obvious meaning. In a later version, the percentage value and it's chart were replaced by a five-colored "red-light", as illustrated in Figure 5.16. Although the red-light provides less information, the later subjects and the earlier ones reviewing it again liked it much better. It was often stated that "the information value [of the red-light] is not very high, but it provides a quick overview without reading all the question answers".

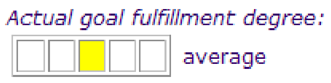


Figure 5.16: The goal fulfillment red light

Although all subjects stated that the presented analysis did not completely describe the problems nature and solve it (mainly because the problem was "more complex as assumed"), all subjects admitted that their awareness of the reasons and circumstances of failing builds was increased and that some of them want to change their personal development process based on the analysis data.

Usability and Functionality This subsection summarizes the aspects of the interview regarding usability and functionality.

As the evaluation was performed as guided interviews, many answers and opinions were recorded. Here, the more interesting ones are just listed and briefly explained. The list is not complete as all feedback of seven hours of interviewing would go beyond the scope of this document. The feedback is separated into two parts, starting with positive feedback followed by negative feedback.

Positive feedback:

- *goal oriented*:(5 of 7 subjects) An issue mentioned by nearly all subjects was that they liked the goal-oriented presentation giving a broad overview over a specific problem domain at a single place. Especially the Hackystat team members who are used to collect data they are interested in using various analysis in different places appreciated goal-orientation.

- *high level data*:(4 of 7 subjects) Similar to the last issue, many subjects who were used to work with low-level or even raw data appreciated the presentation of summarizing, high-level data.
- *interactivity*: 4 of 7 subjects enjoyed the interactivity of the analysis as they were able to explore the dataset of the last few month for themselves, examining time spans which looked interesting to them based on the telemetry streams. This refers indirectly to the retrospective process model used during the interview.
- *quick and easy overview*: (4 of 7) Many subjects lauded hackyCGQMs ability to provide a quick overview, especially in contrast to most other analysis in Hackystat. This is mainly due to it's *graphical representation* and the *aggregation of a many different data types*.
- *explanations why a displayed data set is interesting*: (2 of 7 subjects) All of hackyCGQM goals, questions and metrics have a rationale attached, describing their purpose and intention.
- *ability to evolve plans*: (2 of 7) During the interviews, new hypotheses and new questions were developed by the subjects. Some of these were implemented directly between the interviews. This ability to evolve the measurement plans in an agile way specially impressed two subjects.

Negative feedback:

- *easy to get "lost" in the analysis*: (4 of 7) hackyCGQM just displays all questions and their answers on one single web page. This seemed to be confusing for many subjects.
- *not interactive enough*: (4 of 7) Although interactivity was already mentioned as positive feature of hackyCGQM, most subjects directly criticized that they would love to have more interactivity like interactive graphics with drill downs.
- *not flexible enough*: (3 of 7) Several subjects would have liked to "experiment" more with the system. This includes changing parameters and

options of the measurement program directly in the web interface. Also, ad hoc adding of new questions or metrics was mentioned.

- *graphical representation sometimes confusing*: (3 of 7) The graphical representation in the "buildFailures" example was often not optimal, but confusing. This was mainly because of technical limitations which prevented optimized visualization.
- *ugly user interface*: (3 of 7) The design (visual appearance) of the user interface was considered as ugly by several subjects.
- *too much manual interpretation and interaction necessary*: (2 of 7) Although hackyCGQM tries to automate as many aspects of performing a measurement program as possible, there still remains the manual task of interpreting the question answers. Some subjects wished to have a "one glance and be informed"-style presentation of data. This could maybe be realized by more sophisticated goal fulfillment degrees.

Future Usage and Improvement This paragraph summarizes answers regarding long term improvements and future usage scenarios.

The most stated wish for future extensions and improvements of hackyCGQM was the one for more interactivity, mentioned by four out of seven subjects.

Many of the suggestions for increased interactivity were targeted at the user interface where the subjects wished to drill down charts by clicking on them, reorganizing the order of questions or change the scaling of charts.

But the major issues concerning interactivity where targeted for "on-the-fly" adjustment of the cGQM plans by providing sophisticated configuration options or tools supporting the modification of plugins.

Most of the issues raised in the "future usage and improvement" category are collected and in chapter 6.2 "Future work".

5.4.3 Threats to Validity

This evaluation has two major sources for threatening it's validity:

- The subjects participating are all personal acquaintances or colleges of the interviewer and developer of hackyCGQM. A situation like that usually leads to biased results as the subjects subconsciously tend to adjust their answers according to their sympathy for the interviewer.

  In this evaluation, this issues was addressed by not asking questions which require a rating, but concentrating on questions where the subject had to name their issues (see earlier in section 5.4.1). This results in wider, less biased results. But this has also the major disadvantage that the results are not suitable for statistical evaluation.

- This evaluation did not distinguish between the quality of the usage scenario and the quality of the used technology. Therefore, the presented answers describe a kind of mixture between both.

  While analyzing the results, I tried to separate the feedback manually into feedback related with the scenario and feedback related to the technology itself which is, of course, a source of potential adulteration.

Also, a very important aspect of this evaluation is that it focuses on users using an already developed measurement plan. It does not include any aspects of developing new cGQM plugins, implementing executables, deploying plugins to servers or administrating them.

5.4.4 Discussion

The final results of this evaluation is that cGQM and hackyCGQM do have potential to be used sucessfully in real life applications. Most subjects liked to use the provided cGQM plugins, especially the goal-oriented presentation providing a good overview and ability to browse freely through the whole data set (*retrospective process model*). But still, there remains great potential for improvement. This is especially true for the user interface and the plugin development process.

The presented evaluation results lead to some open issues, which will be discussed in section 6.2 as part of future work chapter.

tool	**Phase 1** Planning	**Phase 2** Definition	**Phase 3** Data collection	**Phase 4** Interpretation
GQM Planner 2		good		
CEFRIEL GQM Tool		good	manual	
VTT Metrifame		good	manual	limited
hackyCGQM		limited	automated only	complex

Table 5.3: hackyCGQM compared to other tools

5.5 Evaluation Summary

This section briefly summarized the most important results of this evaluation chapter.

- The usage of cGQM is limited compared with traditionally performed GQM programs due to it's dependency on metrics which can be measured automatically. Around 45% of encountered, published GQM case studies could be performed using an automated cGQM approach.
- During the user evaluation, most subjects stated that they like the automated, goal-oriented approach of hackyCGQM. But there is still potential for future improvements (see "Future Works", 6.2).
- hackyCGQM can indeed provide near-to-no-cost measurement and analysis cycles, making various non-consecutive usage models more expedient.
- Although cGQMs usage is limited compared with traditional GQM applications, it can provide new ways of visualizing and analyzing data which, due to the huge amount of measurement needed, are nearly impossible when manual measurement and analysis is used. The "overview" plugin (see 5.1.3) serves as an example for that.

hackyCGQM can be considered as another, still young but usable tool for supporting GQM based measurement programs. Compared with the other tools introduced in chapter 2.1.6, it can be classified as in table 5.3. The table, as the one introduced in 2.1.6, classifies the tools by their support of the GQM phases (overview over phases see 2.1.4).

Chapter 6

Conclusion

6.1 Summary

During the course of this thesis, a concept for fully automated GQM-based measurement programs called cGQM was developed. A reference implementation, called hackyCGQM which was integrated into Hackystat was developed. Finally, the implementation was sucessfully evaluated and future directions determined.

The created cGQM framework, hackyCGQM, offers the ability of completely automated execution of measurement programs if the used metrics can be measured in an automated manner. This allows a continuous and long term usage modes of the GQM Paradigm and increases the efficiency and effectiveness of certain measurement and improvement programs.

The major drawback of this approach is that only metrics which can be measured automatically and have a sensor implementation available can be used in cGQM based measurement programs.

But if there is a measurement program which can be designed in such a way that automated metrics are sufficient, the benefit of using cGQM are nearly cost-free program executions after initial setup.

6.2 Future Work

This section summarizes open issues for improving or further evaluating cGQM. Most of these issues were developed during the user interviews or I myself felt that they are important.

- Industry evaluation: Until today, cGQM was only deployed and evaluated in academic settings at the CSDL. For learning about the real effects, benefits and drawbacks, the technology has to be implemented in a "real-life" industrial setting with real users and issues.

  The information extractable from such an evaluation would probably be very valuable for further development.

- Integration of manually measured metrics: As discovered in chapter 5.3, hackyCGQM can only adapt 40% of the GQM case studies encountered in literature. This was mainly due to the dependancy on manually measured data.

  By adding the ability for using manual measurement, this percentage could be increased drastically. The advantages and drawbacks of this modification should be subject of further research.

- Implementation of a fully functional email notification system: The actual version of hackyCGQM still relies on "pull"-style analysis. Using an "push" style analysis via email alerts and thus performing *alert based measurement programs* (see 3.5.4) is prepared and theoretically possible, but no actual measurement program was ever implemented using this technology. Evaluating the benefits of this "push" style programs could also create interesting insights.

- Codebase cleanup and tuning: Although the code base of hackyCGQM is of acceptable quality, it should be cleaned and tuned to increase it's readability, expandability and performance. Also, the unit test coverage should be raised from the actual 80% to at least 90%.

- Improved user interface: The user interface was one of the main issues of negative feedback received during the user interviews in 5.4. Improving and tuning it might be one of the easiest, but yet most effective improvement tasks.

- Tutorials and improved documentation : hackyCGQM has a basic documentation, but for actually attracting new developers using and extending it, comprehensive, easy-to-understand and more complete tutorials and documentations are needed.
- User configurable plugins: Another issue raised by subjects during the user interview were more "just-in-time" configuration options for plugins, allowing users to change certain plugin properties during runtime without recompiling and redeploying the plugins. This would allow a more "explorative" usage style than the actual implementation.
- Executable reference implementation repository: One of the major issues preventing or supporting cGQM's success is the ease of creating new plugins. At the actual moment, only few reference *executables* exists which can be reused in new plugins. But by building a large enough repository containing generic, configurable reference implementation for *executables*, creating new plugins can become much easier and cGQM's success more probable.
- Plan fragment repository: A cGQM plan fragment repository containing parts of plugin-describers would complement the *executable* repository.
- "Plugin-builder" framework: The ultimate goal for improving and supporting the plugin creation process would be the "plugin-builder" framework, allowing even unexperienced users to define their own cGQM analysis.
  The "plugin-builder" could consist of a sophisticated user interface, backed by various repositories containing generic implementations of *executables* and cGQM artifacts including their usage contexts and bound experiences. This concept would aggregate and improve the repository approach of the last two issues with the "Experience Factory" paradigm, both integrated into a sophisticated user interface.
- Sophisticated analysis: The actual cGQM sample measurement programs still rely on a considerable degree of human interpretation. They mainly display charts or graphics which are helpful answering the question, but still need to be interpreted. Integrating sophisticated, computer based analysis techniques like neuronal networks, fuzzy logic, reasoning systems, expert

systems or probabilistic models can allow interesting usage scenarios.

6.3 Acknowledgments

I thank Prof. Dr. Philip Johnson and the staff of the Collaborative Software Engineering Laboratory at the University of Hawai'i Manoa for inspiring my work and giving me the ability of being directly on-site with the Hackystat development team and thus allowing the creation of this thesis.

Appendix A

Plugin Describer Schema Documentation

This appendix chapter briefly introduces the XML schema defining and describing the valid construction of *plugin describers* (see 4.2.3).

The XSD schema file provided by Hackystat can be used in a compatible XML editor and thus supporting the development of plugins by verifying written files against the schema specification.

This schema documentation will just introduce interesting tags, groups or types and explain them briefly.

cGQM : *(root type)* The `cGQM` tag is the *plugin describer's* root tag. It

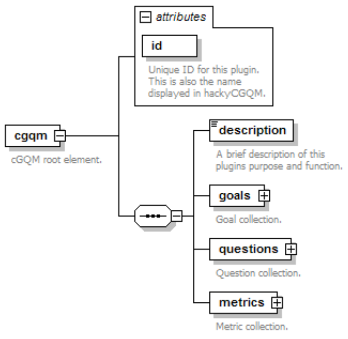

attributes
id
Unique ID for this plugin. This is also the name displayed in hackyCGQM.
cgqm
cGQM root element.
description
A brief description of this plugins purpose and function.
goals
Goal collection.
questions
Question collection.
metrics
Metric collection.

Figure A.1: `cGQM`

specifies the plugin's unique `id` which also serves as it's name in hackyCGQM. In addition to that, the `description` sub-tag contains a String description (can contain HTML when embedded in a CDATA-block).

It also contains the `goal`, `questions` and `metrics` tags which contain, as one might have guessed, the goals, questions and metrics specified by this describer.

cGqmArtifactType : *(complex type)*

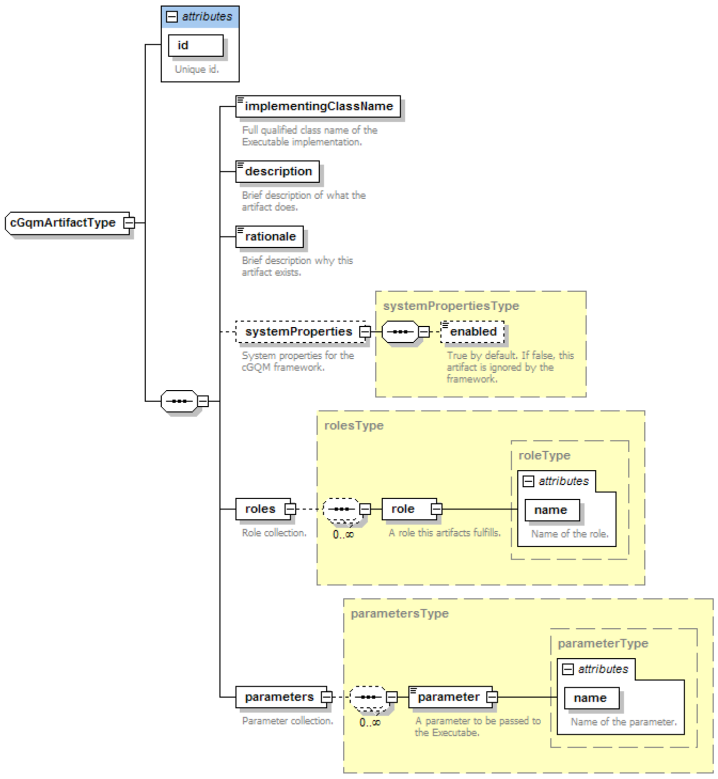

attributes
id
Unique id.
cGqmArtifactType
implementingClassName
Full qualified class name of the Executable implementation.
description
Brief description of what the artifact does.
rationale
Brief description why this artifact exists.
systemProperties
System properties for the cGQM framework.
systemPropertiesType
enabled
True by default. If false, this artifact is ignored by the framework.
rolesType
roles
Role collection.
0..∞
role
A role this artifacts fulfills.
roleType
attributes
name
Name of the role.
parametersType
parameters
Parameter collection.
0..∞
parameter
A parameter to be passed to the Executabe.
parameterType
attributes
name
Name of the parameter.

Figure A.2: `cGqmArtifactType`

The `cGqmArtifactType` type is an abstract base type used as a supertype for the artifact types `goal`, `question` and `metric`.

All the sub-tags and attributes appearing in this type will also appear in the definitions of goals, questions and metrics.

Each artifact is defined by an *unique* id. Also, a `description` and a `rationale` has to be provided. Both of these elements can contain HTML when embedded in CDATA environments. `Description` describes what the artifact does while `rationale` arguments why it is needed.

Each cGQM artifact must have an *executable* (see 4.2.2) assigned, a Java class generating the result. The *executable* is set with the tag `implementingClass-Name`, accepting *fully qualified Java class names* as values.

The `systemProperties` element is optional and contains at the actual moment only the option to disable the whole artifact by setting `enabled` to *false*.

As described in chapter 4.2, each artifact can have multiple roles. These are defined in the `roles` tag.

Also, the *executables* can by parametrized by supplying them with parameters. A `parameter` has a name defined in the according attribute and a string value defined in the tag's body. All parameters are collected in the `parameters` tag.

goalType : *(complex type)*

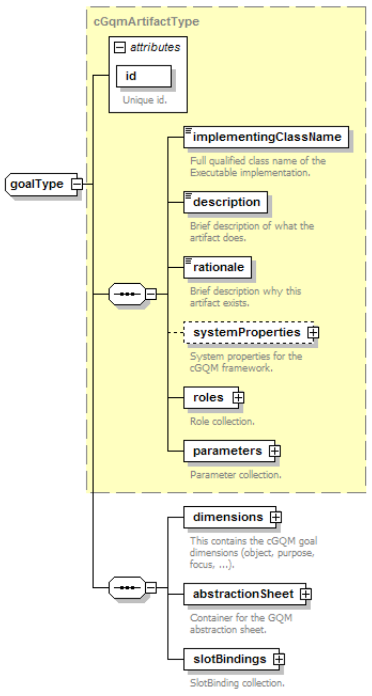

cGqmArtifactType
attributes
id
Unique id.
goalType
implementingClassName
Full qualified class name of the Executable implementation.
description
Brief description of what the artifact does.
rationale
Brief description why this artifact exists.
systemProperties
System properties for the cGQM framework.
roles
Role collection.
parameters
Parameter collection.
dimensions
This contains the cGQM goal dimensions (object, purpose, focus, ...).
abstractionSheet
Container for the GQM abstraction sheet.
slotBindings
SlotBinding collection.

Figure A.3: `goalType`

This complex type is used for the `goal` tags found as sub-tags in `cGQM/goals` and describes a goal. It is derived from `cGqmArtifactType` and inherits all it's tags and attributes.

New elements in `goalType` are `dimensions`, `abstractionSheet` and `slotBindings`.

- `dimensions`: This element defines the *goal dimensions* as described in chapter 2.1.1. Note that this is only used to display the goal dimensions in the web interface. It does not have any further functionality.

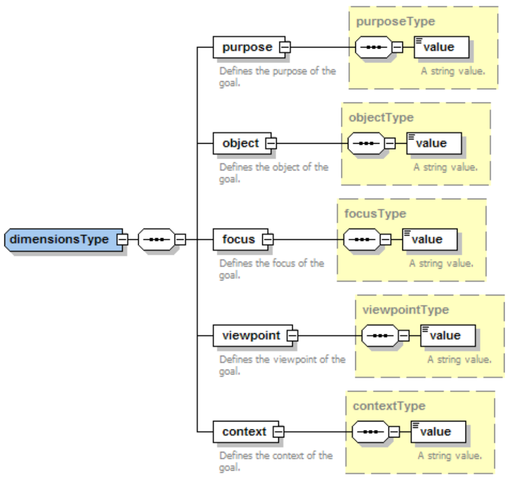


Figure A.4: `goalDimensions`

- `abstractionSheet`: The abstraction sheet, a template for supporting the definition of a GQM goal (see 2.1.5 on page 18), is defined with this element. Each abstraction sheet component can have multiple entries, each defined by a `name` and a `description`. Note that this is only used to display the abstraction sheet in the web interface. It does not have any further functionality.

- `slotBindings`: The concept of slot bindings is described in chapter 4.2.3. In the *plugin describer*, it is realized by defining `slotBinding` elements for each artifact using slots. Each `slotBinding` element binds a object with a given `objectId` to a given `slotName`. The optional `executable` element indicates if this binding just binds an *informative* artifact (value *false*) or one that is used in the *executable* for further calculations (value *true*) (for more information, see 3.4).

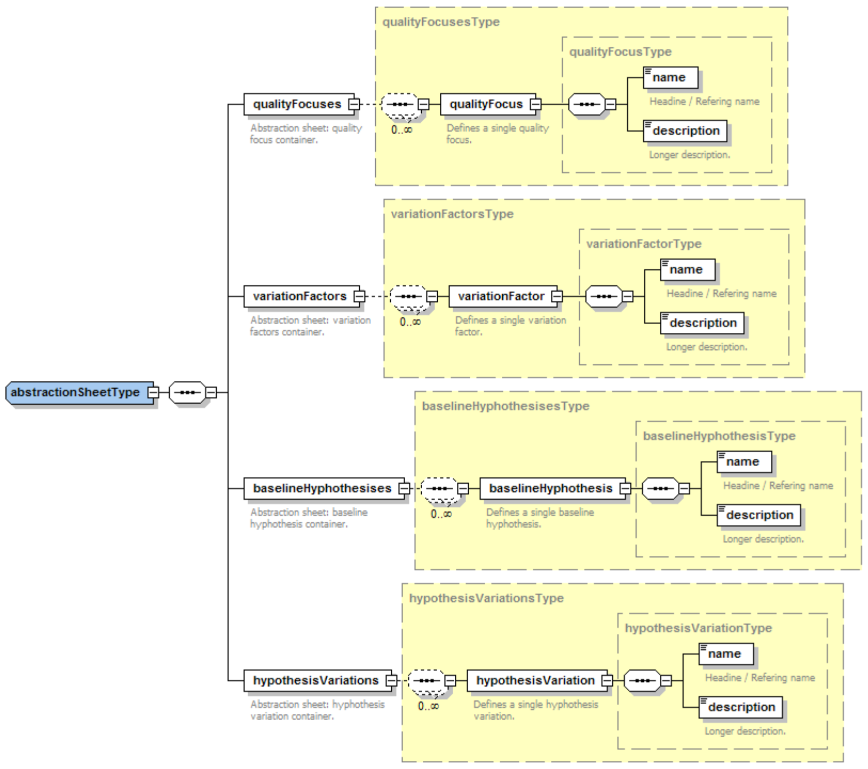


Figure A.5: `abstractionSheet`

questionType : *(complex type)*

This complex type represents a question and is also derived from `cGqmArtifactType`. It extends the base artifact type and is extended with the additional tags `questionText` and `slotBindings`. `slotBindings` has the same function than in `goal type`. `questionText` just contains a string (optionally HTML in CDATA environments) with the question's text.

metricType : *(complex type)*

Similar to `questionType` and `goalType`, `metricType` extends `cGqmArtifactType`. No additional extensions are added.

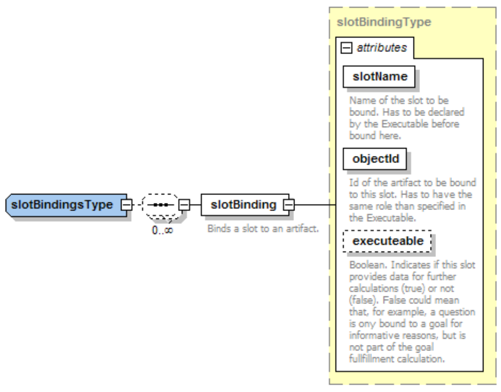

slotBindingsType
0..∞
slotBinding
Binds a slot to an artifact.
slotBindingType
attributes
slotName
Name of the slot to be bound. Has to be declared by the Executable before bound here.
objectId
Id of the artifact to be bound to this slot. Has to have the same role than specified in the Executable.
executeable
Boolean. Indicates if this slot provides data for further calculations (true) or not (false). False could mean that, for example, a question is ony bound to a goal for informative reasons, but is not part of the goal fullfillment calculation.

Figure A.6: `slotBindings`

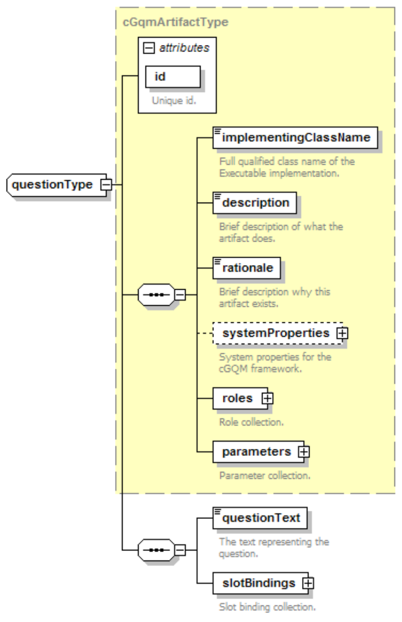

cGqmArtifactType
attributes
id
Unique id.
questionType
implementingClassName
Full qualified class name of the Executable implementation.
description
Brief description of what the artifact does.
rationale
Brief description why this artifact exists.
systemProperties
System properties for the cGQM framework.
roles
Role collection.
parameters
Parameter collection.
questionText
The text representing the question.
slotBindings
Slot binding collection.

Figure A.7: `questionType`

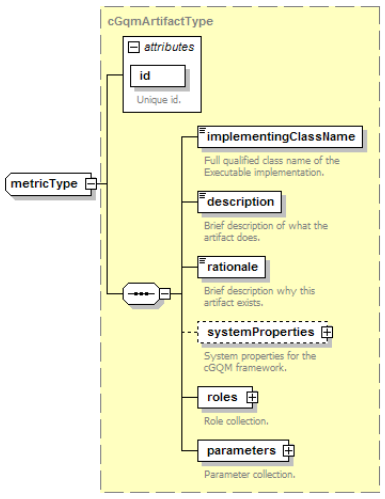

cGqmArtifactType
attributes
id
Unique id.
metricType
implementingClassName
Full qualified class name of the Executable implementation.
description
Brief description of what the artifact does.
rationale
Brief description why this artifact exists.
systemProperties
System properties for the cGQM framework.
roles
Role collection.
parameters
Parameter collection.

Figure A.8: `metricType`

Appendix B

Structural Evaluation - Full result tables

This section contains the full length result tables of the structural evaluation performed in chapter 5.3.

id	pn	tp	note
			Solignen, Berghout; CASE A [8]
G1	0	0	data partially manual
G2	-	-	manual data types
Q1	0	0	data types partially manual
Q2	-	+	data type not supported
Q3	0	0	data types partially manual
Q4	+	+	
Q5	+	+	
Q6	+	+	
Q7	+	+	
Q8	0	+	data type partially not supported
Q9	+	+	
Q10	-	-	data types manually
Q11	-	-	data types manually
Q12	0	+	diagram partially not supported
Q13	+	+	
Q14	+	+	

id	pn	tp	note
Q15	-	-	
Q16	0	+	data types partially not fully supported
Q17	-	0	data type partially not supported, manual data type
Q18	-	-	manual data type
Q19	-	-	manual data type
Q20	-	-	manual data types
Q21	-	-	manual data types
Q22	-	-	manual data types
M1.1	-	-	manual data
M1.2	+	+	
M1.3	+	+	
M2.1	-	+	data type not supported
M3.1	+	+	
M3.2	+	+	
M3.3	+	+	
M3.4	-	-	manual data type
M3.5	0	+	data type partially not supported
M3.6	-	-	data type manual
M3.7	+	+	
M3.8	+	+	
M4.1	+	+	
M5.1	+	+	
M5.2	+	+	
M6.1	+	+	
M7.1	+	+	
M8.1	0	+	data type partially not supported
M9.1	+	+	
M9.2	+	+	
M9.3	+	+	
M9.4	+	+	
M10.1	-	0	data type partially manual
M10.2	-	-	data type manual
M10.3	-	-	data type manual

id	pn	tp	note
M11.1	-	-	data type manual
M11.2	-	-	data type manual
M12.1	+	+	
M12.2	+	+	
M12.3	+	+	
M12.4	+	+	
M12.5	+	+	
M12.6	+	+	
M13.1	+	+	
N13.2	+	+	
M13.3	+	+	
M13.4	+	+	
M13.5	+	+	
M13.6	+	+	
M14.1	+	+	
M14.2	+	+	
M14.3	+	+	
M14.4	+	+	
M14.5	+	+	
M14.6	+	+	
M15.1	-	-	manual data type
M15.2	-	-	manual data type
M16.1	+	+	
M16.2	+	+	
M16.3	0	+	data type not fully supported
M17.1	-	-	manual data type
M17.2	-	-	manual data type
M17.3	-	-	manual data type
M18.1	-	-	manual data type
M18.2	-	-	manual data type
M19.1	-	-	manual data type
M19.2.	-	-	manual data type
M20.1	-	-	manual data type

id	pn	tp	note
M20.2	-	-	manual data type
M21.1	-	-	manual data type
M21.2	-	-	manual data type
M21.3	-	-	manual data type
M21.4	-	-	manual data type
M22.1	-	-	manual data type
M22.2	-	-	manual data type
M22.3	-	-	manual data type
M22.4	+	+	
M22.5	-	-	manual data type
M22.6	-	-	manual data type
			Solignen, Berghout; CASE B [8]
G1	+	+	
Q1.1	+	+	
Q1.2	+	+	
Q1.3	+	+	
Q1.4	+	+	
Q1.5	+	+	
Q1.6	+	+	
Q1.7	+	+	
Q1.8	+	+	
Q1.9	+	+	
Q1.10	+	+	
Q2.1	+	+	
Q2.2	+	+	
Q2.3	+	+	
Q2.4	+	+	
Q2.5	+	+	
Q3.1	0	0	Hard to formalize analysis
Q3.2	0	0	Hard to formalize analysis
Q3.3	0	0	Hard to formalize analysis
Q3.4	0	0	Hard to formalize analysis
Q4.1	-	-	Manual data required

id	**pn**	**tp**	**note**
Q4.2	-	-	Manual data required
Q4.3	-	-	Manual data required
Q4.4	-	-	Manual data required
M.01	+	+	
M.02	+	+	
M.03	+	+	
M.04	+	+	
M.05	+	+	
M.06	+	+	
M.07	+	+	
M.08	+	+	
M.09	+	+	
M.10	+	+	
M.11	+	+	
M.12	+	+	
M.13	+	+	
M.14	-	-	manual data type
M.15	-	-	manual data type
M.16	+	+	
M.17	+	+	
M.18	-	-	manual data type
M.19	-	-	manual data type
M.20	-	-	manual data type
M.21	-	-	manual data type
M.22	-	-	manual data type
M.23	-	-	manual data type
M.24	-	-	manual data type
M.25	+	+	
M.26	+	+	
			Solignen, Berghout; CASE C [8]
G1	-	-	depends on manual data
Q1	-	-	depends on manual data
Q2	-	-	depends on manual data

id	pn	tp	note
Q3	-	-	depends on manual data
Q4	-	-	depends on manual data
Q5	-	-	depends on manual data
Q6	-	-	depends on manual data
M1	-	-	manual data
M2	-	-	manual data
M3	-	-	manual data
M4	-	-	manual data
M5	-	-	manual data
M6	-	-	manual data
M7	-	-	manual data
M8	-	-	manual data
M9	-	-	manual data
M10	-	-	manual data
M11	-	-	manual data
M12	-	-	manual data
M13	-	-	manual data
M14	-	-	manual data
M15	-	-	manual data
M16	-	-	manual data
			Solignen, Berghout; CASE D [8]
G1	-	-	depends on manual data
Q1	-	-	depends on manual data
Q2	-	-	depends on manual data
M1	-	-	manual data
M2	-	-	manual data
M3	-	-	manual data
M4	-	-	manual data
M5	-	-	manual data
M6	-	-	manual data
M7	-	-	manual data
M8	-	-	manual data
M9	-	-	manual data

id	pn	tp	note
M10	-	-	manual data
M11	-	-	manual data
M12	-	-	manual data
M13	-	-	manual data
M14	-	-	manual data
			Fugetta et al. [42]
G1	0	0	partly depended on manual data
G2	+	+	
G3	0	0	partly depended on manual data
G4	+	+	
G5	0	0	partly depended on manual data
Q1	-	-	depends on manual data
Q2	-	-	depends on manual data
Q3	+	+	
Q4	-	-	depends on manual data
Q5	+	+	
Q6	-	-	depends on manual data
Q7	-	-	depends on manual data
Q8	-	-	depends on manual data
Q9	-	-	depends on manual data
Q10	+	+	
Q11	-	-	depends on manual data
Q12	-	-	depends on manual data
Q13	-	-	depends on manual data
Q14	-	-	depends on manual data
Q15	-	-	depends on manual data
Q16	+	+	
Q17	+	+	
Q18	+	+	
Q19	-	-	depends on manual data
Q20	-	-	depends on manual data
Q21	+	+	
Q22	+	+	

id	pn	tp	note
Q23	+	+	
Q24	-	-	depends on manual data
Q25	-	-	depends on manual data
Q26	+	+	
Q27	-	-	depends on manual data
Q28	+	+	
Q29	+	+	
Q30	+	+	
Q31	-	-	depends on manual data
Q32	-	-	depends on manual data
Q33	-	-	depends on manual data
Q34	-	-	depends on manual data
Q35	-	-	depends on manual data
			Lindstrm, Version 2 [43]
G1	-	0	depends party on manual or unimplemented data
G2	-	-	depends on manual data
G3	-	0	depends party on manual or unimplemented data
G4	+	+	
G5	+	+	
G6	-	-	depends on manual data
Q1	0	0	partly depends on manual data
Q2	-	-	depends on manual data
Q3	0	0	result hard to formalize
Q4	+	+	
Q5	-	-	depends on manual data
Q6	+	+	
Q7	-	+	data type not implemented
Q8	-	-	depends on manual data
Q9	-	+	data type not implemented
Q10	-	-	depends on manual data
Q11	0	0	data pertly manual
Q12	-	-	depends on manual data
Q13	-	+	data type not implemented

id	**pn**	**tp**	**note**
Q14	-	-	depends on manual data
Q15	-	-	depends on manual data
Q16	+	+	
Q17	+	+	
Q18	-	0	data type not implemented
Q19	-	-	depends on manual data
Q20	+	+	
Q21	+	+	
Q22	+	+	
Q23	+	+	
Q24	+	+	
Q25	-	-	depends on manual data
Q26	-	-	depends on manual data
Q27	-	-	depends on manual data
Q28	-	-	depends on manual data
Q29	-	-	depends on manual data
Q30	-	-	depends on manual data
Q31	-	-	depends on manual data
M1	+	+	
M2	-	-	depends on manual data
M3	+	+	
M4	-	-	depends on manual data
M5	-	-	depends on manual data
M6	+	+	
M7	-	-	
M8	+	+	
M9	+	+	
M10	+	+	
M11	+	+	
M12	+	+	
M13	+	+	
M14	-	+	data type not implemented
M15	-	-	depends on manual data

id	pn	tp	note
M16	-	-	depends on manual data
M17	-	+	data type not implemented
M18	-	-	depends on manual data
M19	-	-	depends on manual data
M20	+	+	
M21	+	+	
M22	+	+	
M23	+	+	
M24	-	-	depends on manual data
M25	-	-	depends on manual data
M26	-	-	depends on manual data
M27	-	+	data type not implemented
M28	-	+	data type not implemented
M29	-	+	data type not implemented
M30	-	+	data type not implemented
M31	-	-	depends on manual data
M32	-	-	depends on manual data
M33	-	-	depends on manual data
M34	+	+	
M35	+	+	
M36	+	+	
M37	+	+	
M38	+	+	
M39	-	-	depends on manual data
M40	+	+	
M41	+	+	
M42	+	+	
M43	-	0	not implemented
M44	-	-	depends on manual data
M45	+	+	
M46	+	+	
M47	+	+	
M48	+	+	

id	**pn**	**tp**	**note**
M49	-	0	not implemented
M50	+	+	
M51	+	+	
M52	-	-	depends on manual data
M53	-	-	depends on manual data
M54	-	-	depends on manual data
M55	-	-	depends on manual data
M56	-	-	depends on manual data
M57	+	+	
M58	-	-	depends on manual data
M59	-	-	depends on manual data
M60	-	-	depends on manual data
M61	-	-	depends on manual data
M62	-	-	depends on manual data
M63	-	-	depends on manual data
M64	-	-	depends on manual data
M65	-	-	depends on manual data
M66	-	-	depends on manual data
M67	-	-	depends on manual data
M68	-	-	depends on manual data

Table B.1: Results of the structural mapping

Appendix C

User Interview Question Guidelines

This appendix chapter contains the questionnaire used in the user interview evaluation in chapter 5.4. The questionnaire has to be understood as rough guideline as the interviews were designed to be flexible enough to concentrate on interesting aspects. This means, that not all questions were used in each interview and that some interviews concentrated on questions now even part of the questionnaire.

Questionnaire

1. **The Build Failure Example**

    1.1 Did this analysis give you additional insight into the build process, our/your development behavior or the reasons behind failing builds? (yes/no)

    1.2 Was the long term analysis with the full 5 month period or the 5 shorter analysis more useful to you? Which one would you abandon if you could just use one?

    1.3 What were the most valuable insights you got from this analysis?

    1.4 Did this data confirm or disprove existing suspicions you had about the failing builds? If yes, which ones?

- 1.5 How much of these information did you know (not just suspected) before? What was it? From which source did you get your old information?
- 1.6 After seeing all this, do you see the need to change your development behaviour? If yes, what would you change?
- 1.7 If you would change your behavior, do think hackyCGQM is able to show that this change has an effect? If no, why?
- 1.8 Can you think about additional questions interesting in the context of the Build Failure Example? If yes, which ones?
- 1.9 Can state additional hypothesis for the build failure problem?
- 1.10 Which important aspects of the build failure scenario are not covered by this analysis?

2. **Usability**
   - 2.1 What are the major benefits of this presentation of data compared to other Hackystat technologies?
   - 2.2 What are the major disadvantages of this presentation of data compared to other Hackystat technologies?
   - 2.3 What are the major usability problems you had with hackyCGQM?
   - 2.4 What aspects of hackyCGQM did you like?
   - 2.5 What aspects of hackyCGQM did you not like (beside those you already mentioned in question 3)
   - 2.6 What would be your major argument for using hackyCGQM?
   - 2.7 What would be your major argument against using hackyCGQM?
3. **Future Usage**
   - 3.1 Can you come up with interesting aspects of software development which could be examined with hackyCGQM? (if so, please list them)
   - 3.2 Which cool new feature of hackyCGQM would you love to get?
4. **Miscellaneous**

4.1 Any additional comments?

4.2 Anything you want to tell me?

4.3 Anything else you can think of?

Appendix D

Indices

List of Figures

List of Tables

Acronyms

GQM Goal/ Question/ Metric Paradigm

cGQM continuous Goal/ Question/ Metric Paradigm

CSDL Collaborative Software Engineering Institute, University of Hawai'i

QIP Quality Improvement Paradigm

SPCC Software Project Control Center

PSP Personal Software Process

SOAP Simple Object Access Protocol

CMM Capability Maturity Model

SEI Software Engineering Institute

ISO International Standard Organization

SPICE Software Process Improvement and Capability dEtermination

ISO9001 International Standard for Quality management systems

XML eXtensible Meta Language

XSD XML Schema Definition

XSLT eXtensible Stylesheet Language Transformation

GPL General Public License

UML Unified Modeling Language

JSP Java Server Pages

IDE Integrated Development Environment

Bibliography

[1] P.M. Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications, Nashville, TN, December 2001.

[2] V.R. Basili. Software modeling and measurement: The goal question metric paradigm. Technical Report CS-TR-2956, University of Maryland, College Park, 1992.

[3] D.M. Weiss V.R. Basili. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[4] T. DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.

[5] Encyclopedia britannica. ⟨`http://www.britannica.com//`⟩.

[6] Wikipedia. ⟨`http://www.wikipedia.org//`⟩.

[7] Pfleeger Fenton. *Software Metrics A Rigorous and PracticalApproach*. Chapman and Hall, London, 1997.

[8] E. Berghout R. van Solingen. *The Goal/Question/Metric Method: A practical guide for quality improvement of software deleopment*. McGraw-Hill Int., London, 1999.

[9] R.S. Kaplan. Measures for manufacturing excellence. *Harvard Business School Press*, 1990.

[10] Johnson and Jim. Chaos: The dollar drain of it project failures. *Application Development Trends*, 1:41–47, 1995.

[11] W. C. Peterson. Seis software process program - presentation to the board of visitors. Technical report, Software Engineering Institute, Carnegie Mellon University, 1997.

[12] K Hyde and D Wilson. Intangible benefits of cmm-based software process improvement. *Software Process Improvement and Practice*, 2004.

[13] L. H. Putnam and M. C. Mah. Software by the numbers: An aerial view of the software metrics landscape. *American Programmer*, October 1998.

[14] V.R. Basili and G. Caldiera. Improve software quality by reusing knowledge and experience. *Sloan Management Review, MIT Press*, Fall, 1995.

[15] W. S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[16] W. S. Humphrey. Characterizing the software process. *IEEE Software*, 5(2), 1988.

[17] Quality management systems: Iso 9001. International Organization for Standardization, ISO/IEC ISO 9001:2000.

[18] Spice: Iso 15504. International Organization for Standardization, ISO/IEC 155504-1:2004.

[19] Anne M. Disney and Philip M. Johnson. Investigating data quality problems in the psp. *ACM SIGSOFT Software Engineering Notes archive*, 6:143 – 152, November 1998.

[20] Herbsleb, Zubrow, Goldenson, Hayer, and Paulk. Software quality and the capability maturity model. *Communications of the ACM*, 1997.

[21] C.M. Lott C. Differding, B. Hoisl. Technology package for the goal question metric paradigm. Technical Report 281, Computer Science Department, Technical University Kaiserslautern, Kaiserslautern, Germany, 1996.

[22] H.D. Rombach L.C. Briand, C.M. Differding. Practical guidelines for measurement-based process improvement. *Software Process - Improvement and Practice*, 2(2):253–280, 1996.

[23] J. Nielsen. Usability engineering at a discount. *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, 1989.

[24] VTT. Metriflame, 1999.

[25] L. Lavazza. Providing automated support for the gqm measurement process. *IEEE Software*, pages 56–62, May 2000.

[26] C. Differding. *Adaptive Measurement Plans for Software Development*. Phd thesis, Institute of Experimental Software Engineering Kaiserslautern, 2001.

[27] Quality characteristics and guidelines for their use, iso 9126. International Organization for Standardization, ISO/IEC ISO 9126.

[28] Christiane Gresse, Barbara Hoisl, and Jürgen Wüst. A process model for planning GQM-based measurement. Technical Report STTI-95-04-E, University of Kaiserslautern, 67653 Kaiserslautern, Germany, 1995.

[29] J. Heidrich J. Muench. Software project control centers: Concepts and approaches. Technical report, Fraunhofer Institute of Experimental Software Engineering, 2003.

[30] J. Heidrich. Effective data interpretation and presentation in software projects. Technical report, Computer Science Department, Technical University Kaiserslautern, 2003.

[31] J. Heidrich. Custom-made visualization for software project control. Technical report, Computer Science Department, Technical University Kaiserslautern, 2003.

[32] W. S. Humphrey. *Managing the Software Engineering*. Addison-Wesley, 1989.

[33] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[34] W3C Soap web site. ⟨`http://www.w3.org/TR/soap/`⟩.

[35] Apache Jakarta Tomcat web site. ⟨`http://jakarta.apache.org/tomcat/`⟩.

[36] Johnson, Kou, Paulding, Zhang, Kagawa, and Yamashita. Improving software development management through software project telemetry. Technical Report CSDL-03-13, Collaborative Software Engineering Laboratory, University of Hawaii, 2004.

[37] Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. ADDISON-WESLEY LONGMAN, AMSTERDAM, 1999.

[38] Sun javabeans specification. ⟨`http://java.sun.com/products/javabeans/reference/api/index.html`⟩.

[39] Jibx binding framework. ⟨`http://jibx.sourceforge.net/`⟩.

[40] Checkstyle - coding style checker. ⟨`http://checkstyle.sourceforge.net//`⟩.

[41] Junit - unit testing for java. ⟨`http://www.junit.org//`⟩.

[42] A. Fuggetta, L. Lavazza, and S. Morasca. Applying gqm in an industrial software factory. *ACM Transactions on Software Engineering and Methology*, 7(4):441–448, October 1998.

[43] B. Linstrm. A software measurement case study using gqm. Technical Report LUTEDX(TETS-5522)1-72(2004), Lund Institute of Technology, 2004.