Part Five: IT Project Management

10. Software Quality And Defect Management

Software projects (as part of IT projects) result in a piece of code which can be

- directly installed executed on a particular Operating System (OS) platform (like Windows, UNIX, MacOS) or can
- delivered in source code, requiring compilation and linking of the respective programs prior of execution.

Regarding quality, the software has to fulfil particular requirements:

1. Usability: The software as to fulfil the defined tasks and the published use cases ('works as designed').
2. Conformance: The software has to comply to the published functional aspects ('works as expected').
3. Absence of bugs: The software should be reasonable bug-free ('works under all circumstances').
4. Security: The software should neither directly nor indirectly impact the security context of the user ('works without security impact'), except where explicitly stated.
5. Performance: If performance is not part of the Usability, the software should use as little system resources as possible and achieve maximum performance ('works with little system impact and high performance').

These goals can only be achieved

- by means of a qualified software design, which fits to the respective tasks and considers the requirements,
- with a well-suited OS and perhaps necessary middleware,
- developing the software in a (quality) controlled environment with supporting infrastructure, and
- by aid of a qualified defect tracking and documentation system.

On the other hand, the software quality is independent of the programming language itself, and hence whether the code is actually executed via an interpreter (script, macro), as byte-code (within a virtual machine), or directly as binary (including OS loader statements). Of course, the choice of the programming language has a substantial impact on the performance (and occasional on security too). Figure 66 tries to outline the conflicting attributes in software developments.

The chain:

Design/Planning -> Coding/Development -> Control/Improvement

is typically limited by budget and time-to-market conditions. Shortages in any of those steps has a direct influx on the quality of the software product.

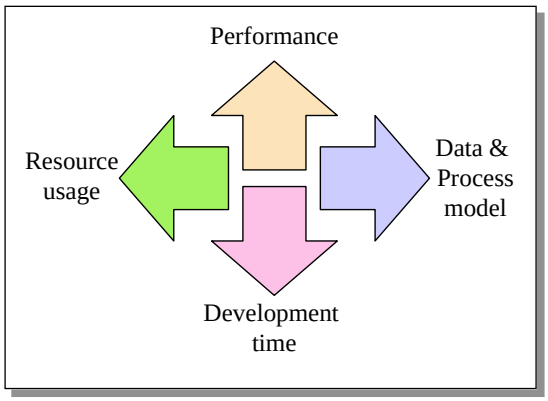

Figure 66: Conflicting attributes in software development

10.1 Software Quality Management According To ISO 9000

The standard ISO 9000-1 questions software quality for 'Information Systems" IS in chapter A.3:

- [Management attention] Is there any IT Manager for the IS accountable and responsible to define requirements and to approve changes ?
- [Quality Management System] Are all requirements for the IS explicitly laid down in documents ?
- [Audits] Are all requirements accompanied by a description how to verify it's conformance ?

According to ISO 9000 high-level management attention the most important factor required for quality management. The quality management system QMS itself can be viewed from an (a) descriptive and (b) from an operational persective as outlined in figure 67.

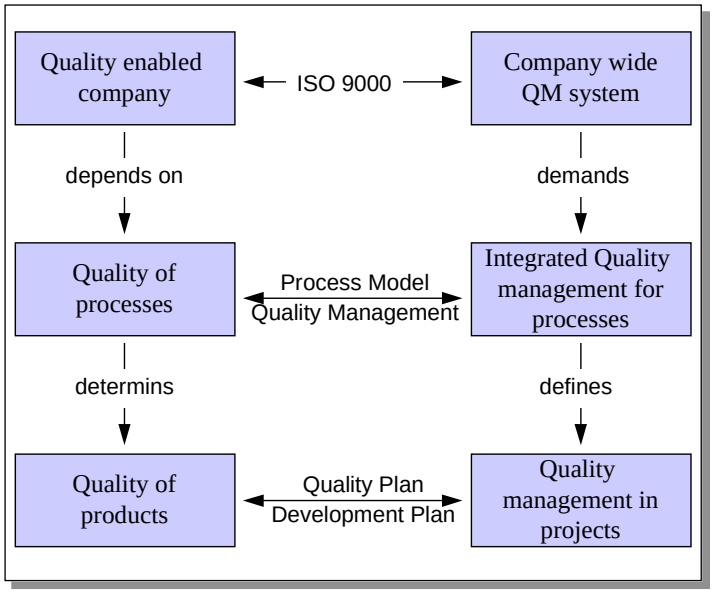

Figure 67: Set-up of a Quality Management System according to ISO 9000-3 [~ Wallmüller1995]

The content of a QMS depends on the production branch described in the following ISO standards:

- ISO 9001: Standard for quality management in design, development, production, assembly, and customer services.
- ISO 9002: Standard for quality management for production and assembly.
- ISO 9003: Standard for quality management for final testing and control.

Thus for software development, ISO 9001 is the required standard and demands continuous process improvements as shown in figure 68:

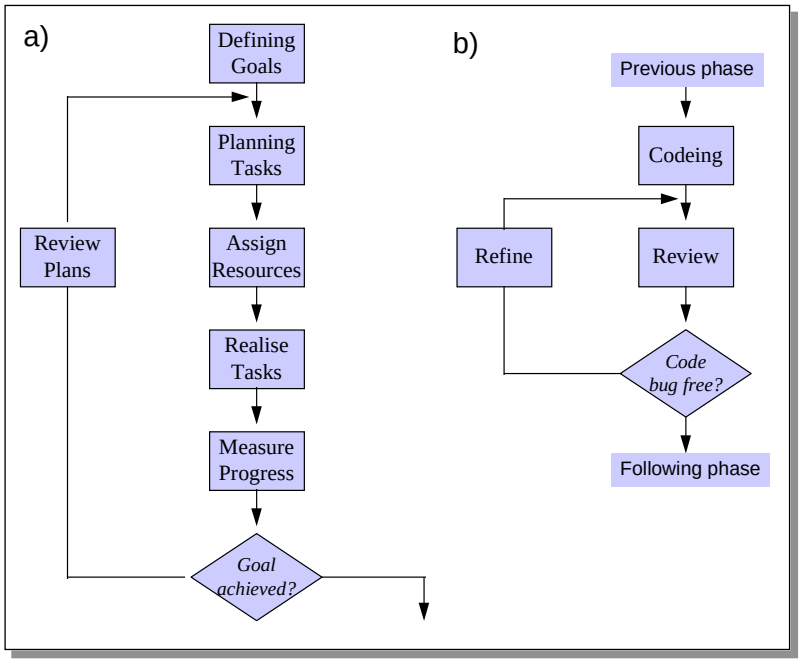

Figure 68: Quality Management for software development
a) iterative process improvements regarding quality
b) constant improvements for developments by means of testing

The standard ISO 10013 provides in addition a lay-out of a Quality Management System (QMS) and emphasises the rôle of documentation in order to achieve conformance with this standard (figure 69).

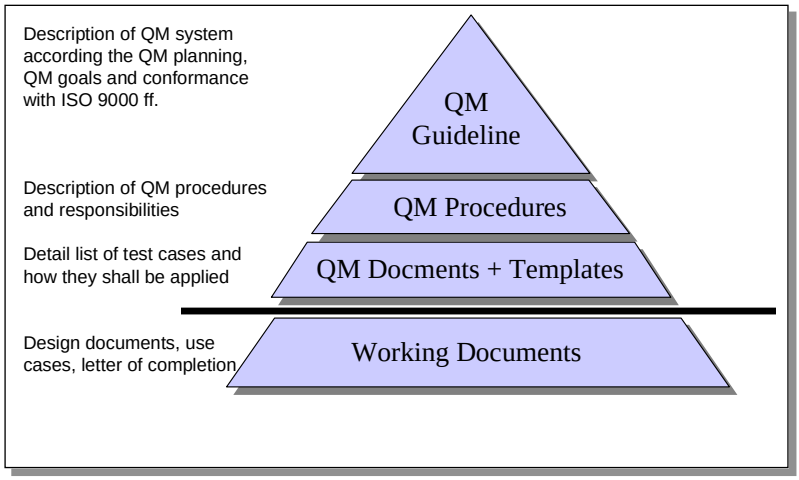

Figure 68: Hierarchy of a Quality Management System according to ISO 100013 [~ Patresch1998]

While setting up a QMS according to ISO 9001, in fact one has to include QM elements from the ISO standards 9001, 9002, and 9003. Thus, the final QMS handbook has to have the following scope:

Chapter	Title	Requirements		
		ISO 9001	ISO 9002	ISO 9003
1.	Responsibility of Managment	complete	complete	partially
2.	Quality Management System	complete	complete	partially
3.	Contract Verification	complete	complete	complete
4.	Design Management	complete	not available	not available
5.	Management of documents and code	complete	complete	complete
6.	Ordering Management	complete	complete	not available
7.	Management of third party deliveries	complete	complete	complete
8.	Labeling and Tracking deliveries	complete	complete	partially
9.	Process Management	complete	complete	not available
10.	Test procedures	complete	complete	partially
11.	Verification of test tools	complete	complete	complete
12.	Status of tests	complete	complete	complete
13.	Management of missing products	complete	complete	partially
14.	Correction and preventive means	complete	complete	partially
15.	Usability, packaging, and deployment	complete	complete	complete
16.	Management of quality reports	complete	complete	partially
17.	Internal quality audits	complete	complete	partially
18.	Education and training	complete	complete	partially
19.	Maintenance and customer service	complete	complete	not available
20.	Statistical means and procedures	complete	complete	partially

Table 3: Elements of a QMS document

Apart from assuring the quality management cycle for products, the conformance of the existing QMS has to be verified by means of *Audits* (ISO 9000-1). Audits are part of the ISO 9000 certification chain and may include:

- Content and conformance of the QMS handbook with the ISO standards.
- Operational conformance with the QMS handbook.
- Tests of the product quality.
- Capability of the project team to manage QM processes.

In Germany, some major organisation support the certification according to ISO 9000:

- Deutsche Gesellschaft zur Zeritifierung von Qualitätssicherungssystemen (DQS)
- Deutsche Gesellschaft für Qualität (DGQ)
- Technischer Überwachungs Verein (TÜV)

10.2 Use-Cases And Test-Cases

In today's software development, the design phase of the product or any component of the product includes a *Use Case*. The Use Case is a functional description of way, the product or the component is supposed to work.

The event of Object Oriented (OO) programming has produced a certain schematic description of the components action flow, known as *Unified Model Language UML* which can be considered of a standardized description. The UML introduces several sets of charts:

General	Mainly Programming
Use Cases Charts	Class Charts
Paketing Charts	Object Charts
Usage- and Deployment Charts	Communication Charts
State Charts	Time Charts
Activity Charts	Interaction Charts
Component Charts	Sequence Charts

Table 4: UML Chart types [Oesterreich1998]

UML Use Case charts are the de-facto standard to represent dependencies as can be picked up from figure 69.

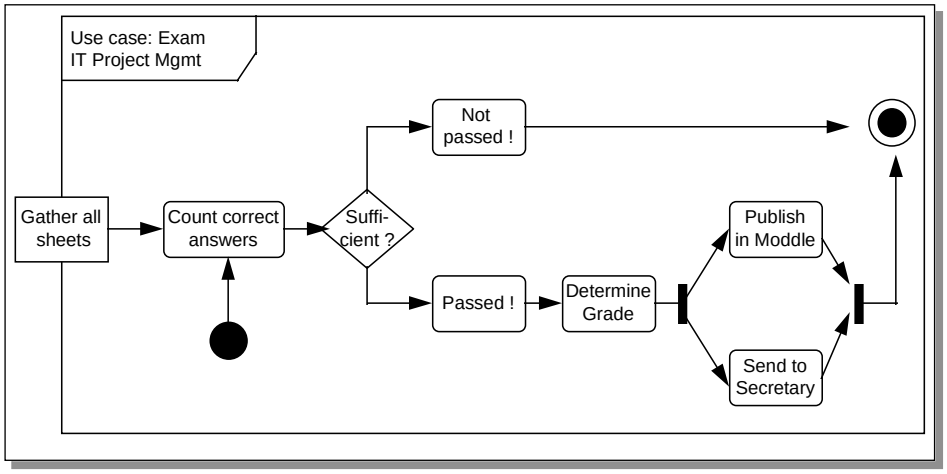

Figure 69: Use Case how to evaluate a written exam [~ Wikipedia]

Another common method is to specify the Use Case in an *Activity Table* while using a generic template (table 4).

Regarding software development each component, as described in terms of a Work Unit (PMBoK), has to be accompanied by a Use Case. Thus, the Use Case is part of the software design/development.

According to the Quality Plan (QP) the QA department has to create *Test Case* suites. Here, the functional dependencies of the component are used to derive tests whether the software component is in conformance with the Use Case or not. Hence, any substantial tests have to prepared in term of Test Cases, which in turn depend on the existence of qualified Use Cases. In particular, to derive the necessary Use and Test Cases becomes difficult and time-consuming for complex software products. As a result, complexity and quality of software products are believed to be opposing attributes.

<Name of the Use Case>	**<Version>**	**<Date>**	
Author	My Name		
Brief Description			
Primary Actor			
Secondary Actor			
Trigger, Predecessor			
Event, Successor			
Technical dependencies			
Opent items, Remarks			
Step	**Actor**	**Activity**	**Junctions**
1.			
1.1			
1.2			
2.			
Exception			
Variant			
Option			

Table 4: Template for an Activity Table of a Use Case

As a result, only in the case of a well-documented software component, it's quality can be measured. A poorly documented software is almost impossible to gauge and effectively leads to a frustrated user (and tester !), or a user who does use only parts of the software's capability (while paid for the total). In spite of this, pretty often the QA department sets up tests based on (educated) guesses or needs to do a reverse engineering of the software component (however, typically inhibited by most companies policies).

A Test Case should include the following tests (starting from the Use Case):

- *Default behaviour*: Provides the software component the expected results for default settings and input variables ?

- *Conditional behaviour*: In case the component offers additional 'switches'/'options' and/or 'arguments'/'parameters', do they work as expected (described) ?
- *Extreme behaviour*: How does the component recognise input values out of specification and what are the results ?
- *Erratic behaviour*: How does the component react, in case the required environmental conditions are not met/outside specification ?

The qualified Test Case would detail the expected results based on a functional breakdown for the software component, while explicitly mentioning the conditions of the individual tests.

In conclusion, these dependencies very much underline the importance of qualified design and development documentation. In particular, it is required to provide an exhaustive list of error and return codes for any software component.

Since any documentation is expansive, in particular for development documents two approaches are common to ease this task for the developers:

- *In line documentation within the code*: In the software code itself, relevant sections are documented with a specific mark-up language easy to parse and to collect. The programming language PERL has pioneered this, known as "Plain Old Documentation" POD.
- *Documentation within the development framework*: Some frameworks, like Eclipse, use context-sensitive information retrieval by means of a plug-in. Here, the current changes can be recognised and documented and in the same token saved (and retrieved) in a version-dependent manner.

10.3 Defect Management

As outlined,

> *a defect is a deviation of a software component from the documented behaviour and/or expected output.*

10.3.1 Attributes of a Defect

For any defect, the following attributes can be assigned

Source

The reason for a Defect could be:

- *A programming error*; commonly known as *Bug*.
- *A programming context error*; the anticipated (software) functionality works differently as documented/expected.
- *A design error*; the programmer realised the code accordingly to the design, but this was inappropriate for the task.
- *A documentation error*; the software component reacts differently with respect to the documented behaviour.

Priority

In our today's understanding, we use the following *Priorities* for the *defect fixing*:

- *Undefined (-1)*: Showstopper; the defect needs to be fixed because it inhibits any further testing or any potential use of the software component.
- *One (1)*: The software component is inadequate or any use.
- *Two (2)*: The software includes severe deficiencies and may lead to substantial deviations from the expectations.
- *Three (3)*: The software shows deficiencies but they can be compensated (by work around).
- *Four (4)*: Less-relevant errors have been accounted, which are not important for general use.
- *Five (5)*: Errors have been accounted, but are not mainly due to insufficient documentation.

Category

Any Defect is assigned to a component *Category*. Occasionally, a software module may include several components:

- User Interface (Input / Output)
- Middleware, Transport Layer, Interfaces/APIs
- Back-end (e.g. Database)

It is the responsibility of the software design to attach a certain component to category, suitable for Defect tracking.

Ownership

A defect is assigned for error-fixing to a particular *Owner*. In turn, every software is developed and perhaps trigger by a certain person/team/organisation/company. Thus, there should be a relationship between *Ownership* of the defect and the *Authorship* of the software component.

Typically, *Revision Control Systems* (*RCS*) and/or *Integrated Development Environment* (*IDE*) will automatically insert Author information taken from the user environment in terms of a header.

Versions

Versionising the software component can be done explicitly by the developer or is automatically added by the RCS/IDE. Independent of the components version, it typically is developed for and available in a certain *Release* which provides a numerical (or verbal) identification of the whole project.

Project

A major software component or a set of components is typically identified as *Project*. Projects maybe subdivided in subprojects. However, this depends on the WBS (in PMBoK terms) and can be freely chosen in any software development project.

State

The state describes the recognition of the defect (whether it is new, open or closed etc.) and it's assignment state, as discussed in the defect life cycle.

Due-Date

The expected date, when the defect shall be fixed (this depends on priority).

10.3.2 Defect Lifecycle

For software development, essential part of the QM system is a bug-tracking or Defect Management software. One of the most-common systems is *Bugzilla* (public domain). Typically, any Defect Management system uses a database as back-end and a graphical (ie. Web based) front-end.

Logically, such a system allows us to define a *Lifecycle* for a defect. This Lifecycle can be forged to our own needs, or follows a standard procedure. In practice, companies may want to use one common system for Defect management and as well for Incident and Problem management, since the Lifecycle idea is the same. Such *Trouble Ticket Systems* allow a *Class* definition for the occurred error to be reported:

- *Defects* - Software Development, Bug tracking
- *Incidents* - Deviations for a defined process (erratically behaviour); (none-) recurring
- *Problems* - Set of (inter-depending) incidents with (known) common source

In terms of Incidents and Problems not the *Priority* is of importance but rather it's *Severity*. Here, the correct *routing* (= assignment) of tickets is most relevant. Such systems may be extended by an *Artificial Intelligence AI* component with allows correlation of incidents, by time, location, and/or source. In this way, incidents may automatically sorted and forwarded and solutions strategies are proposed based on similar incidents or known (and solved) problems.

The *State* of a defect will be changed and reflects the current actions on this defect. A minimal scheme could be the following:

Defect State	Meaning
Unconfirmed	The defect has been reported but has not been checked successfully
New	The defect has been reported, but yet not assigned and/or verified
Assigned	The defect has been assigned and forwarded to a responsible developer
Reopened	The defect was handled and closed, but requires further investigation/treatments
Resolved	The defect has be solved and the solution requires approval
Verified	The defect is solved and the solution was verified
Closed	The defect is solved and the solution is integrated

Table 5: Status of a defect according to Bugzilla

In addition, it might be necessary to qualify the solution (or in software development terms 'the fix') in some more detail. Bugzilla uses the following approach:

Fix State	Meaning
Fixed	A bug-fix has been applied
Invalid	The defect was no due to a bug; further information is required
Wontfix	The defect can't be solved under the current conditions (time, budget)
Later	The fix for the defect is deferred (next release/version/update)
Remind	No fix will be provided now, but considered for a forthcoming version
Duplicate	The defect is a duplicate of another one
Worksforme	The defect can not be reproduced in the developer's environment

Table 6: Status of a defect fixing according to Bugzilla

Transition changes among phases are not allowed to happen arbitrarily, but rather depend on the life-cycle model in place. This can be expressed in complex D*efect State Charts* as shown in figure 70:

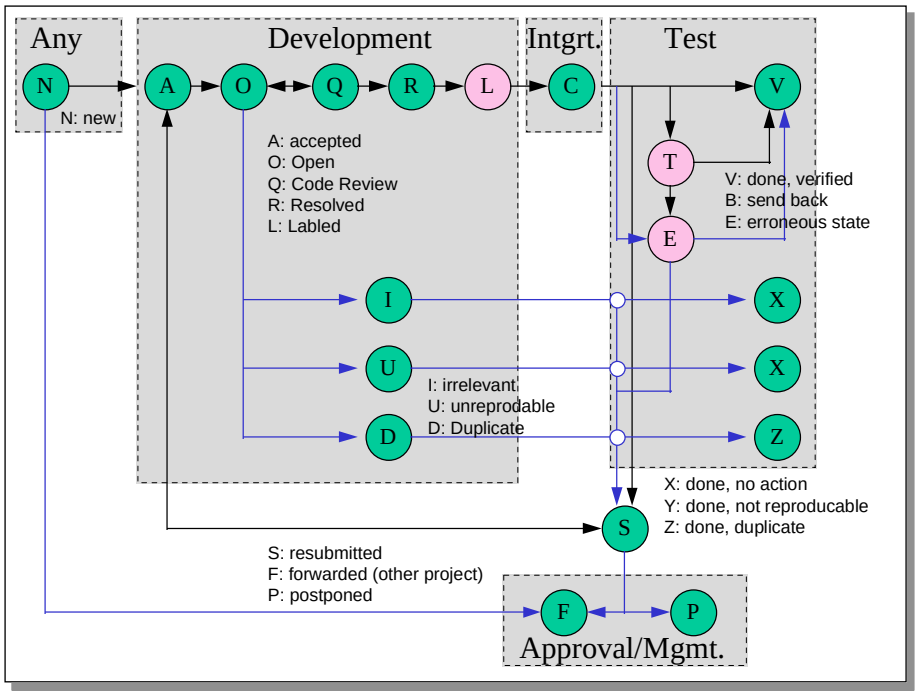

Figure 70: State chart for defects [~ T-Mobile]

10.5 QA Reports

Quality Management or Quality Assignment Reports are generated regularly (for instance every week) and includes essentially tow different reports:

- a *statistical report*, representing the defects per project/subproject in terms of priority and component
- an *individual report*, focusing on the most important defects, while provide a short description of its current state and forthcoming solution strategies and due-dates.

Typically, a statistical reports is shown in terms of 'lego charts' allowing a quick understanding of the project's defect distribution in terms of priority and component (figure 71). In order to allow a quality measure, the following analysis shall be done:

1. How many defects of priority X have been fixed since last report ?
2. How many new defects of priority X have open since then ?

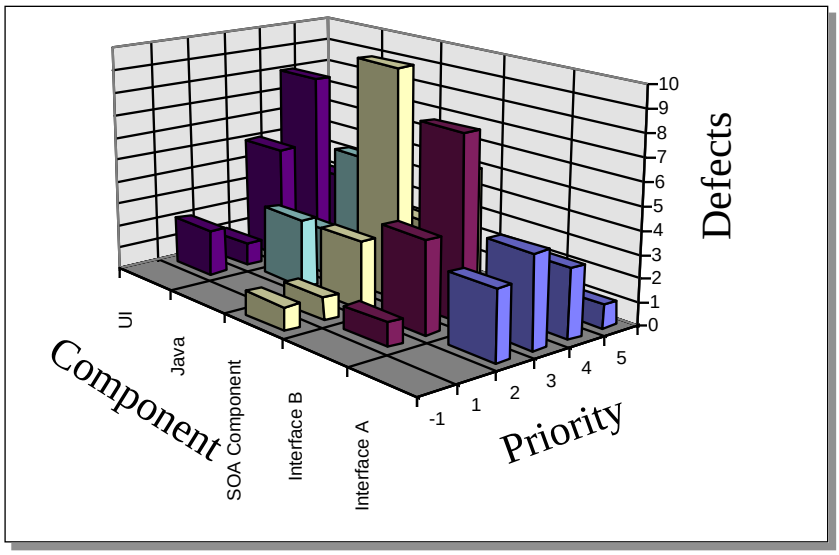

Figure 71: Defect distribution in terms of components and priorities

One essential task of the Quality Manager is to gauge individual events and bring them to attention. The following reasons could be considered:

- *Showstopper*: The defect impacts all other tests, since they depend on it's fix.
- *Criticality*: The defect inhibits the use of an important component and the solution is critical for the whole project.
- *Due-Date*: The solution for this defect has be postponed too often and disturbs other developments.

Together with the head of development, the Quality Manager will re-assess the defects and either require an intensified consideration of the defect or perhaps re-prioterise it. The head of development will then re-assign and re-schedule developers to potentially fix the defect.

10.6 Estimating Remaining Defects

In particular in the advent of a forthcoming release, it is important to have a quantitative estimate of the number of potential open bugs in the product or perhaps per component.

In case of qualified Quality Management system and under the assumption that defects have been treated in a controlled manner, we already have the following QA information:

- Distribution of the number of new defects in terms of priority and components.
- Distribution the number of fixed defects in dependency of priority and component.

What we don't currently have, is

- the number and distribution of unknown defects.

It is most common for software development to allow within a 'release' a number less severe defects. This in turn requires an estimate of the number of potentially remaining defects which have to be added to the number of known bugs.

The key here is to use additionally development information:

- Phase 1: In the beginning of the software project, the established code base is small and bugs (even prio 1) happen often.
- Phase 2: While the development team becomes familiar with it's tools, the approach, writing a set common utility programs or classes, and gathering more and more experience, development becomes streamlined and the code base grows proportional in time and in numbers of developers. Watching this from the Quality Manager's perspective, quality increases and defects 'come and go'.
- Phase 3: However, coming close to a scheduled release, often development realises that completion is behind the original schedule. Thus, missing (but promised) functionalities have to be included in a rush. The code basis will probably increase significantly.

Figure 72 shows a sample, where the development plan assumes code completion happens accordingly to gaussian distribution, while real coding is deferred by some δ in time.

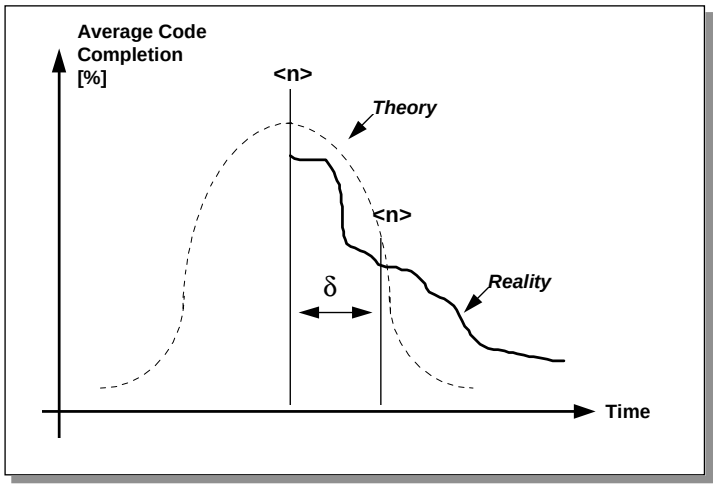

Figure 72: Average code completion as scheduled (dashed), as realised (solid)

In order to control quality and to consider the rapidly growing code base, the Quality Manager has to correlate the number of defects with the actual checked in code.

For any software development it expected, that the number of defects depends on the lines of code produced, expressed as 'Defects/kLoc' (number of defects per 1000 lines of code). Thus, even with most advanced QA means a certain number of bugs is be present and is acceptable. The *Quality Management Plan* (QMP) will probably detail, what is the amount of acceptable bugs and will provide a threshold (figure 73).

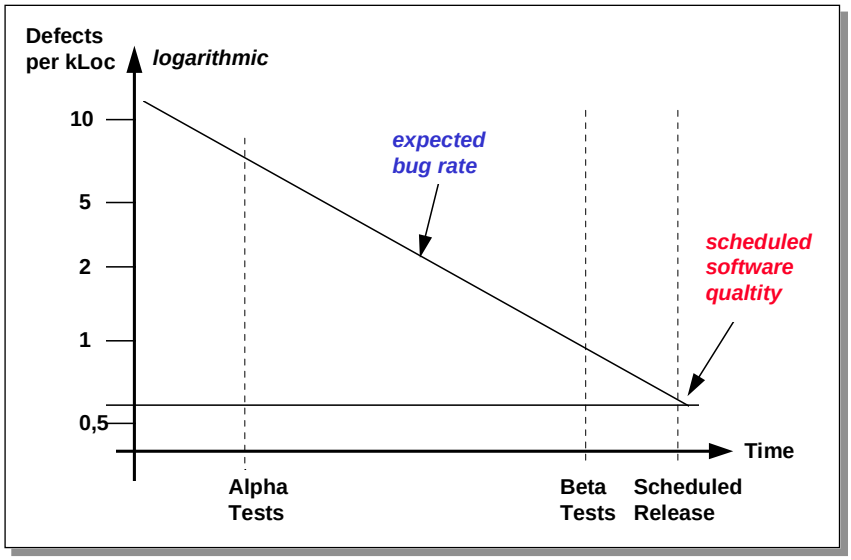

Figure 73: QMP chart with an estimate of acceptable defects for the scheduled release

Now, it is task of the Quality Manger to correlate the number of identified, fixed defects per kLoc and show this distribution on the same time line. A qualified extrapolating (not necessarily linear) will yield a guess of the number unidentified as shown in figure 74:

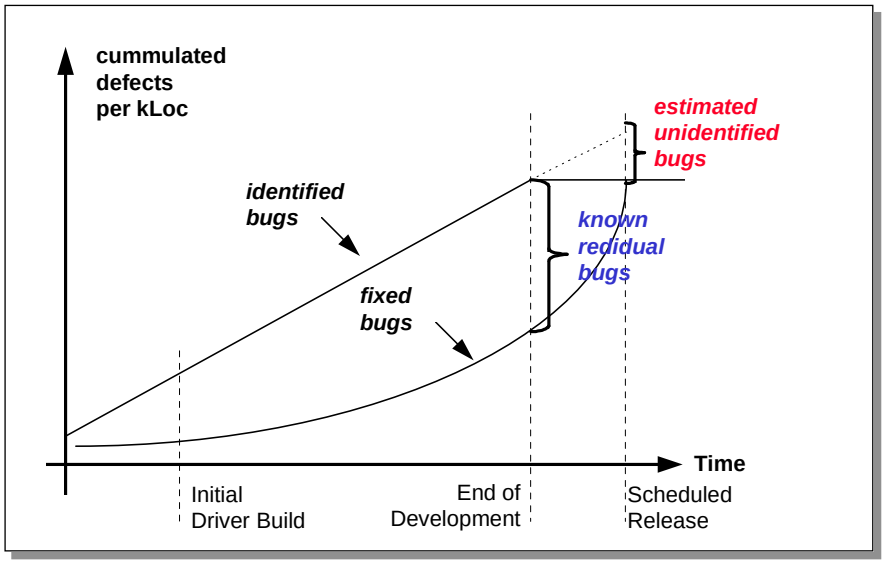

Figure 74: Estimation of unidentified defects for the final release

10.7 Development And Testing Environments

In order to support the software quality management cycle, as part of the QMP different environments are usually used and referred to. Now, what is a environment ?

> *If common resources, including operating system, middleware (database), and other required applications a dedicated and configured for a dedicated task, we can refer his as environment.*

Typically, the following environments are used and configured for larger software developments:

- [DEV] *Development environments*: This includes in particular any required compiler and linker, Integrated Development Frameworks, and Source Code Control Systems. At least one specific development environment is required for *Release Management*.
- [UAT] *User Acceptance Test environments*: The (system) test environment's reflect closely, but on a smaller scale the later production environment; specific tools for software tests and quality management may be available. Here, the final quality approval will be carried out.
- [INT] *Integration Test environments*: In case third party systems need to be included, one particular UAT can be used as Integration Test environment.
- [REF] *Reference environments*: This environment should be sized and configured comparable to the production environment. Here, performance and regression tests can be performed. Occasionally, the reference environments can be used as backup and fail-over systems for the production.
- [PROD] *Production environments*: Full sized production environment. Different from UAT, additional hardening could have taken place. Their impact has to be figured out in the reference environment.

References

[Wallmüller1995] Ganzheitliches Qualitätsmanagement in der Informationsverarbeitung, Carl Hanser Verlag, München 1995

[Petresch1998] Einführung in das Software-Qualitätsmanagement, Logs Verlag, Berlin 1998

[Oesterreich1998] Objectorientierte Software Entwicklung, Oldenbourg Verlag, München Wien 1998