Measuring code inspections and feeding information back to project members

Harri Kemppainen

University of Tampere
Department of Computer and
Information Sciences
Master's thesis
April 2002

University of Tampere
Department of Computer and Information Sciences
Kemppainen Harri: Measuring code inspections and feeding information back to project members.
Master's Thesis, 51 Pages
April 2002

Software inspections have several commonly agreed measures like the number of defects, duration, effort, preparation time etc. These measures are commonly collected and used to analyse the inspection process. These measures are rarely related to individuals who work in the project. To bridge this cap we define five roles and an information need for each role. All roles and information needs are related to code inspection.

From an information need we derive measures and indicators to feed some of the collected information back to project members.

We use a measurement information model to describe the path from information need to measures and indicators. We use measures, which are commonly collected from code inspections.

Keywords: Software Measurement, Software Metrics, and Measurement Information Model.

Table of contents

1 Introduction

Software is used increasingly even in common household equipment. You can hardly find a car, a TV or a VCR set without a processor and embedded software. Software size and complexity increases all the time. Increased usage of software should lead to better user interfaces and improved functionality. All these improvements are void if the quality of the software is lagging behind.

Quality is perhaps the most important aspect of software engineering. And software measurement is very important tool for software engineering. Software measurement can be used to measure quality, but also to measure how changes in working methods affect to quality. Software inspection is a good example of an activity that improves quality.

Software inspection was introduced by Fagan [1976]. It is a formal method of co-workers reviewing colleagues' work to find problems in it. Inspections have been used to review all documents created in software projects. Removing problems from documents (including code) improves communication and prevents problems seeding new problems in later phases of software development. Removing problems before the end user sees them in the final product is of course important. Inspections detect problems at the same phase as they are introduced making it a cost effective method to remove problems.

Literature very often promotes the usage of software inspection by reporting savings and improvements to software development. For daily work of individuals, savings or improvements may not have any direct effect. Therefore measures collected from inspections may have no direct meaning.

Personal Software Process [Humphrey, 1997] introduces inspection with personal measures and personal perception and experience that inspections make creation of software easier. This raises a question if other participants of software project could also get useful and direct information from inspection. This can be seen also as a short feedback loop.

To answer the question and to create short feedback loops, we will define five generic code inspection related roles (Chapter 5). For each role, we give one

information need that presents role's personal interest towards software inspection. Information need presents just one aspect of role's interests. Information needs also bridge target of the interest to measurement and presentations of measurements. It should also ensure that what is measured also answers to information need. We will use our own examples to illustrate our ideas.

In Chapters 2, 3 and 4 we go through essential concepts of software engineering (Chapter 2), software inspection (Chapter 3) and software measurement (Chapter 4). Chapter 4 also introduces the most important concept of this work - measurement information model. Chapter 5 introduces roles and Chapter 6 information needs for each role.

Software engineering, software measurement and software inspection are covered only briefly in this work. There are though excellent textbooks on software engineering (Pfleeger[2000] or Pressmann[2000]), software measurement (Fenton and Pfleeger[1999]) and software inspection (Gilb and Graham[1993] or Strauss and Ebenau[1993]) should anyone want to get more thorough information.

2. Software Engineering

Boehm [1981, pp 16] defines software engineering as "the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation".

For "the application of science and mathematics" we could use numbers or measures. Thus, measurement as a method of acquiring measures is an essential part of software engineering. Software engineering is the framework and software measurement is an extremely important tool.

Computer equipment has to be "useful to man". To be useful it has to fulfil the users' needs. The software product has to be good enough and it has to be easy to use. It's important to notice that the software engineering covers not just the actual program ran on the computer but also the documentation and work procedures that are related to the use of the computer system. Software engineers have to take into account also the environment where computer systems are used.

Pfleeger [1998, pp 5] emphasises the quality of the work and the work products. "Any Hacker can write code to make something work, but it takes the skill and understanding of a professional software engineer to produce code that is robust, easy to understand and maintain, and does its job in the most efficient and effective way possible. Consequently, software engineering is about designing and developing high-quality software." Quality is not explicitly mentioned in Boehm's definition, but to be "useful to man" the computer system has to be of a good quality.

2.1 Software process

The software process is a collection of activities to create software products. The process may be documented, well defined or there may be no documentation at all. As projects grow larger, well-defined process gets more important. It is possible to do good quality work in small-scale project without any process description. In larger projects, work has to be divided to several phases and to

several developers. A documented and well-defined process guides how division of work can be done and ensures that each team works in a similar manner. It also reminds to take all the necessary steps in the development process. Documentation makes it easier to repeat and follow a successful development process. It's also easier to make changes to process or evaluate process, when it is documented. Software measurement should be an integral part of processes.

The software process is often modelled with a lifecycle. The model of a software process helps us to understand what is happening in the software development. It also limits our views. If we base our development on waterfall lifecycle (Figure 2.1) then our measures and understanding of the software process will reflect that structure. For instance, measurement of the effort is probably easiest to conduct as time used in each phase.

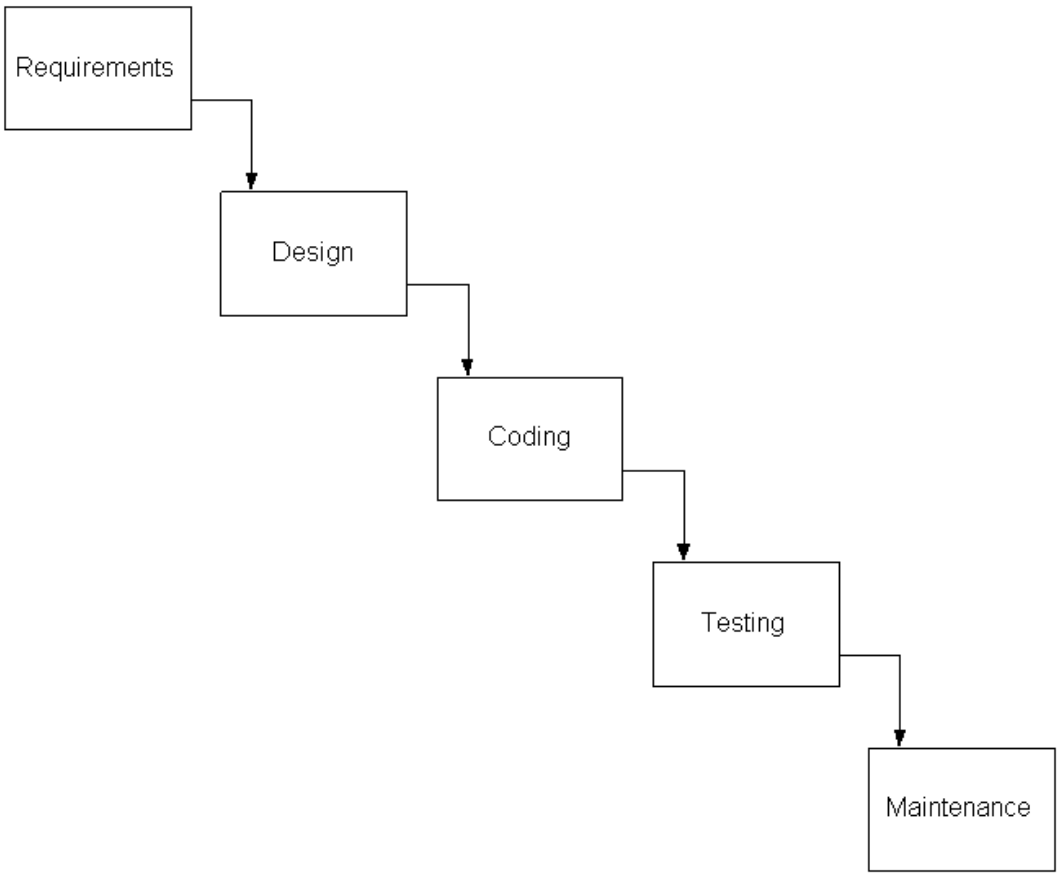

Figure 2.1. Simplified waterfall model

We will use the simplified waterfall lifecycle presented in Figure 2.1 to model the software process. We assume that the code is based on a design, the design is based on a set of requirements and there will be testing (and other activities) after coding. It means that for each piece of code there is a relevant design (document) and a set of requirements. Tested software is configured for certain use (and perhaps tested again) and packaged for delivery. We will use the simplified waterfall model where testing is separated as own activity and maintenance represents all activities after testing. Almost any model where these four phases (requirements, design, coding, testing) are present in set order could be used instead of our simplified waterfall model (for more on models and lifecycle see [Pfleeger, 1999, pp 44-58]).

Requirements describe what kind of software product the project team is going to implement. Requirements can be seen as an agreement between the customer and the project team. A customer is not always available for requirement definition, so marketing or other data is used to guess what customers want. Requirements have to be detailed enough, so that all the important issues are covered. If something important is omitted, it will be noticed later and changes later in the lifecycle are much more expensive. Requirements cannot be too detailed, because it limits available options. Optimally requirements describe the software product unambiguously, containing all necessary features and constraints, but without limiting possible implementations. Requirements should not contain design.

The design can be seen as an iteration from requirements. The design tells how larger product is split into smaller pieces that are more manageable. Design contains description of all pieces and how they interact with each other, so that product will behave as described in requirements. Interfaces are a way to describe interaction between pieces of the software product.

Design is the basis for coding. The code is a realisation of interfaces and internal structures of each piece of software product. Coding is not just transformation of the design. If it were it could be done automatically. There is still a lot of work in implementation; the design is always an incomplete

description of what the code is supposed to do. An excellent set of requirements and design can be ruined with sloppy implementation. On the other hand, it is unlikely that bad requirements and design will lead to outstanding quality in the software product.

Testing is used to both validate that the implementation does what is defined in requirements and to find failures. Testing may have several steps as pieces of software product are integrated together into larger subsystems. After a software product is tested it will be packaged and released. Packaging may include configuring the software for certain environment or use.

Maintenance contains all activities after a product is (sold and) delivered to customer. Customer may find failures or after some use may want changes to product. Maintenance ends when the product is no longer used (or when further support is denied).

2.2 Error, Fault and Failure

It is human to err. That is also true with programmers. Very often problems with computers or programs are called bugs or errors, but computers or programs do not err; they do exactly what they are told to do. In software engineering bugs are called failures. Failure is a situation where the product does not behave, as it should. The behaviour of the product is described in requirements and documentation. Users usually do not have access to requirements so failures are problems in the behaviour against the documentation or against how the user expects the product to behave.

A failure means a fault somewhere in the instructions for computer. The fault may be a result from faulty design or faulty requirements. There are many reasons why a work product may fail, as there are many reasons for faults.

Humans make errors, not computers. Occasionally humans do errors when they do programming, documentation or design. In code, document or design, such an *error* is called a *fault*. Sometimes faults demonstrate their existence during the execution of the code; such incident is a *failure*. One error can cause several faults. Designer makes an error and creates a fault to the design. A fault

in the design may lead to several faults in the code. One fault may cause software to failure in several ways. One failure may be a result of several faults.

It's important to notice that the real requirements may differ from the documented requirements. Earlier we defined a failure as a problem in the behaviour against what the user expects. In such a case product may behave as the requirement document specifies, but the requirement document does not describe the behaviour as the user expects it. This is a fault in the requirement document. The fault in the requirement specification is then copied to the design and from the design to the code and finally causes the product to fail. One big source of faults is changes to requirement documents.

2.3 Software Quality

Most commonly, software quality is seen as absence of failures. Of course, it's important that software doesn't crash, but there are other aspects of quality. For end user such aspects are reliability, security and efficiency (software is fast enough and does not waste resources). On the other hand, if these things are important they could be recorded in requirements.

From the developer's point of view, aspects of quality are reusability, maintainability, testability, understandability, flexibility, portability or interoperability. These are all related to the software development and maintenance. All these aspects are also internal, in the sense that the user will not see their existence, but may see their effect to the software development.

In this work, our view of quality is very narrow; absence of faults and how code inspections can be used to detect (and remove) them. We also assume that removing faults (from code with inspections) will remove failures from software product.

3. Software Inspection

Software inspection is an important tool in quality software development. In this chapter, we will see how inspections relate to other activities of software development. We will also study how inspections are conducted. Our view of inspections will be generic and not very detailed.

Fagan [1976] introduced software inspections at late seventies. Inspection is a method for finding faults in software work products. An inspection is usually conducted at the same phase as the work product is created. So, a requirement document is inspected at requirements phase, before design is created from requirements.

Formal inspections include a (formal) meeting and a well-defined process in addition to just checking documents for faults. A chairman leads the meeting and all found faults are recorded. There is a named activity for each participant (chairman, secretary, author and checker). Faults can be found by comparing work product to source documents (for example code to design or design to requirements), using checklists or just by personal expertise. Checklist is collection of commonly found problems. It is used as a remainder or guideline on things that should be checked.

Inspections are an effective method to find faults in written documents. Any document created during software development can be inspected (plans, requirements, the design, the architecture, test plans, the user documentation, the code etc.). Figure 3.1 shows how inspections fit in the simplified waterfall lifecycle. Inspections can be positioned equally into other software lifecycle models. If lifecycle is based on phases then inspections are held at the same phase, possibly ending the phase.

Inspections cover quite a large collection of similar reviewing methods (walkthrough, peer review etc.). We will concentrate on formal code inspections (from now on formal inspection or just inspection).

Usually findings of the inspection are called defects. The difference between defect and fault is very vague. To emphasise the division between faults and failures, the term defect is not used in this work. In a way, it is a redundant

term. The most of inspection and fault related terms from this work can be found from literature by replacing fault with defect, for example fault density is usually defect density. In this work fault is used instead to keep terminology simple.

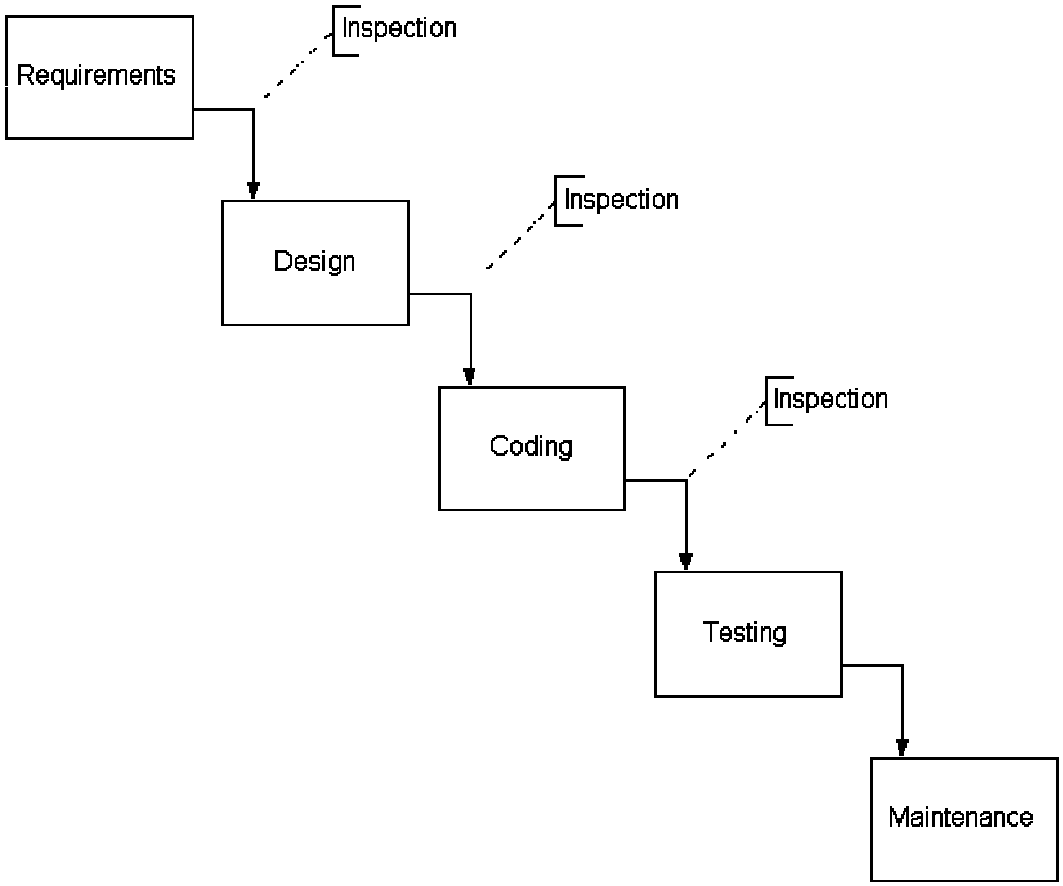

Figure 3.1 Simplified waterfall model with inspections.

Figure 3.2 shows common phases or activities of the inspection. The preparation sets up an inspection process (schedule, participants, documents etc.). The individual checking is perhaps the most important part of an inspection. In checking phase inspectors (participants of an inspection) go individually through documents to find faults. Faults are collected and recorded at the meeting. After the meeting the author fixes faults and at follow-up fixes are confirmed.

At the preparation inspectors agree on a schedule and get all the needed documents for the individual checking. Inspectors may also get a specific role for checking. Roles of inspection relate to certain activities or interest groups, like users or testing. Role "user" means that the inspector goes through the document trying to see how the user would see it. Inspector with role "testing" would concentrate on aspects that affect on the testing and so on.

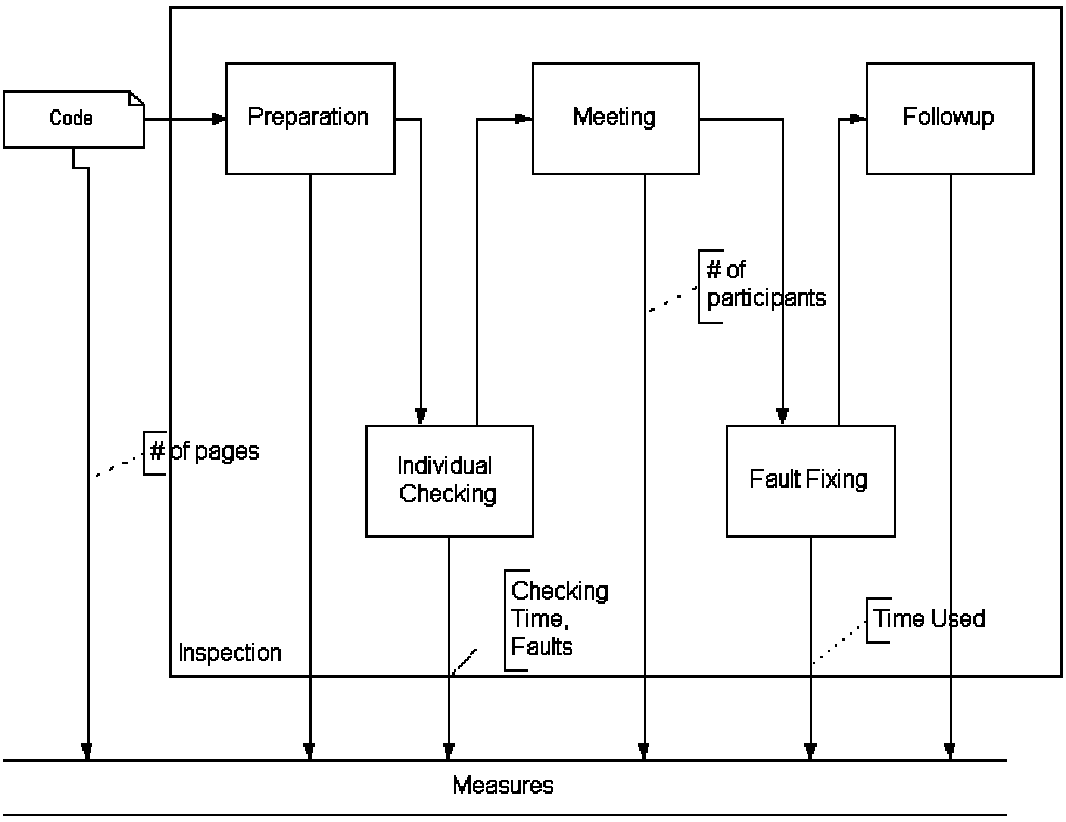

Figure 3.2 The inspection process with example measures.

All inspectors gather to a meeting to collect their findings after the individual checking. In the meeting it is possible to discuss is finding really a fault. Although meeting should not stretch too long, so discussion should be kept to the point. It is mainly chairman's responsibility to keep meetings short and focused.

From the meeting, the author gets a list of faults. She corrects the faults from inspected documents. The chairman or someone named at meeting follows-up

that faults are fixed. And after the follow-up the inspection process is finished. At the follow up chairman (or other named person) checks also that all measures have been recorded. It is important to notice that inspectors look for faults that might cause the final product to fail. Spelling errors are faults, but they rarely cause failures.

An inspection may find faults in source documents too. The design may show weaknesses in the requirements. The author of the inspected document does not fix faults in source documents. They are delivered to the authors of the source documents. Changes to source documents may cause delays to the current document, but its best to fix faults as early as possible. Worst choice would be to ignore the fault and let it propagate to implementation and final product. Should source documents be changed then? Yes, they should. A source document may be a source to several other documents. For example, suppose a fault in an interface, which is found in code inspection and the interface is fixed in code but not in the design document. Others who use the same interface don't know from the design that there is a fault and the implementation has changed in one part of the code.

Inspection is a method to improve quality by finding (and removing) faults. Not all faults cause failures, but it is much easier to fix faults than to fix failures. Fault is an insiders' view to a problem and if a developer knows that there is a fault, he usually knows exactly where it is. Once you know where the fault is it is easy and fast to fix it. Failure is a problem in the product behaviour. Debugging and other methods are used to find the faults that caused the failure and it takes time. Once assumed faults are found, the code has to be fixed, compiled and packaged. If the newly build product behaves correctly, we can assume that the problem was fixed.

Based on Boehm's [1981, pp 40] data it takes ten times more effort to fix a fault from code than fixing the original fault in a requirements document. It takes also ten times more effort to fix fault after release than at coding phase.

3.1 Inspections as a project management tool

Inspections should be used from the very start of the project. Requirements are inspected so that we can find faults before they cause real problems in design and implementation. Inspecting design against requirements assures that design adheres to requirements. Inspecting code against design ensures that what was originally set in requirements and then further refined in the design is properly implemented in code.

Each step from requirements to implementation adds more detail. Inspection is a tool that keeps this flow of information coherent. Each step adds more details and inspections can be used to track that all documents are in line. It is easy to see that if one step of lifecycle is omitted or is not inspected, the chain from requirements to implementation is broken and code inspections lose some of their strength.

As abstract ideas from requirements are refined through lifecycle to code, each inspection is a learning opportunity. Inspectors see how requirements can be developed to design and design to code. They see several solutions and they see some that fail. Design may reveal that requirements were not good enough or implementation may reveal weaknesses in design. Inspection can teach how to create better requirements for design and better design for code. Also the author gets feedback from more detailed levels on what they could not have anticipated or what works and what didn't work.

3.2 Controlling inspections

Inspections take lots of time. Total duration can be long (preparation, checking, meeting, fixing and follow-up) and several participants checking and meeting means lots of effort. It is quite natural that inspections have to be kept effective and under control. It has been already mentioned that inspection is probably the most effective tool to find faults, but what does it actually mean.

Cost of a fault gets higher as it propagates through the lifecycle. Inspection is one tool to find these faults as early as possible. In some cases already at the same phase as the fault was introduced. Based on the data Boehm [1981] has presented, letting a fault to slip to the next phase makes fixing costs rise several

times higher. So using a couple of hours per fault to find and fix them in inspection is acceptable compared to much higher costs to fix the fault later. Of course, inspections have to be efficient. There is no sense to use hundred hours per fault in inspection, since it may be less costly to find and fix the fault by other means

The inspection effectiveness is a percentage of faults that the inspection finds from the document [Gilb and Graham, 1993, pp 437]. Code inspections can find up to 60% of faults [Gilb and Graham, 1993, pp106]. Efficiency tells how much effort is spent to discover a fault [Gilb and Graham, 1993, pp 437]. Efficiency can be calculated for an individual checker, the inspection team or the whole project. To get the best out of the inspections, both efficiency and effectiveness should be as high as possible.

Duration from the start of the inspection to the end of the inspection may get long. Just arranging the inspection meetings for several inspectors may be hard and may postpone ending of the inspection for several days. Porter and Votta [1997] have conducted tests for inspections with meeting and without meeting (faults are collected with a web based collaboration tools). As you could assume, an inspection without meeting took significantly shorter calendar time. Porter and Votta [1997] has also tested the effect of inspection meeting and claims that they are useless and actually decrease efficiency. Land, Sauer and Jeffery [1997] have reported that inspection meetings actually increase efficiency by decreasing the amount of false alarms. These perceptions seem contradictory, but they do make sense. Porter and Votta [1997] studied only the amount of findings in inspections, where Land, Sauer and Jeffery studied also how many of the findings were actually faults. Where Porter sees decrease in efficiency (as there are less reported findings) Land, Sauer and Jeffery see an improvement in efficiency as there is a significant decrease of false alarms and the author can concentrate on fixing real problems. Inspections have been used in software industry for over 20 years, but there are still lots of things we don't know.

4 Software Measurement

In a narrow sense, software measurement is about measuring various aspects of software development to get insight to it. In a large sense software measurement covers theory and practice behind the measurement of the software, analysis of the measures, definition of measures and validation of measures.

The software measurement is essential part of the software engineering. Without measures, accurate and scientific approach to the software engineering is impossible. In practice, measures are an invaluable part of the software development. All kinds of measures are used in the software development although they are not called software measures. There are deadlines or schedules, errors are tracked and fixed. And usually someone wants to know who is working in the project, so that salaries can be paid. The difference is that the software measure is specifically collected to guide and improve the software development. Software measures are collected with a standard procedure, so that each measure is comparable.

The software measurement has three sources of measures: Resources, processes and work products. Resources can be the staff assigned to project, computers, office environment or clerical services for project etc. The effort (time) staff use in the project is an example of an important resource measure. A work product is any document created in the project. In addition to the code, such documents are the user documentation, the design, the requirements and test plans. The size in pages or lines of code is an example of a (work) product measure. The inspection efficiency is an example of a process measure.

Measures are tools for managing the project. Measures help to tell what is going on in the project, is it on the schedule or is there a problem that has to be dealt with. Measures can be used to plan and estimate for example the duration of the project or amount of faults. Process improvement activities can be guided by measures and measures can be used to evaluate benefits of the changes in the software process.

Measures are useful to all project members. A software engineer can use measures to manage and guide her work. For example, the size of a code segment is estimated to be 100 lines of code and it's due in five days, so the work pace should be about 20 lines per day. If she makes 40 lines of code at the first day, project manager can let her go early on Friday, since she will probably finish in time.

The software measurement is a continuous process. As measures give more visibility to the target, it may be possible to redefine measures more clearly and precisely. As measures improve, they also give a clearer picture of the target. In a sense measures reflect how well the target is known. It's like going to an unfamiliar area without a map. At start, we have very vague understanding of our surroundings. As we move around, we learn geographical features. As we get familiar with our new surroundings, we know where to find things, but also we know the fastest routes between places. If we have a clock, we can try to validate which route is actually fastest.

4.1. Measurement theory

Measure is a mapping (homomorphism or representational theory) from an *attribute* of an *entity* of the real world to a value (in mathematical world). An entity of the real world is any object or a thing we are interested about. Examples of entities are code, inspection process, or staff. An attribute of an entity is any property or quality that the entity possesses. Code has such attributes as size, complexity, etc. Staff has such attributes as effort, salary, skills etc.

Mapping from attributes to measures has to conserve relationships that attributes have in the real world. If we can say that one unit of code is twice as big as another unit of code, then the measures of size have to have the same relationship. Relationships can be reduced to five scale types: nominal, ordinal, interval, ratio and absolute. These scales are used in statistics to define what statistical operations can be done. [Fenton and Pfleeger, 1999 and Zuse, 1998]

Operations that can be done increase from nominal to absolute. Scales are ordered from the most restrictive (nominal) to the most relaxed scale (absolute). Each later scale type allows all operations of the previous scale.

We map attribute characteristics to a numerical value. We can then analyse those values with statistical methods. Available methods depend on scale type our measure presents. From our analysis we draw conclusions and project them back to the real world.

4.2. Nominal scale

The nominal scale presents the most primitive form of measurement, a simple classification. Examples of such a classification are numerous: modifier of a function (private, public etc), type of inspected document (requirement, design, code, test plan, etc.) and so on. Classes are equal, i.e. there is no order between them, and any numeric or symbolic representation can be a measure. For example, a measure for type of inspected document could be A for requirement, B for design, C for code etc. In this case, letters are in order but they don't have to be. Measures could be 82 for Requirements, 68 for Design, 67 for Code and 84 for Test plan (measure is ASCII value of the first character. should be easy to handle in a computer program, but hard for user to remember).

4.3. Ordinal scale

The ordinal scale is like the nominal scale, except that classes have an order. The order between classes gives us more options in analysis. An example of ordinal scale is phase of development (in waterfall model). Phases are ordered, design phase follows requirements phase, coding follows design and so forth. Phases are equal and it makes no sense to add one phase to another phase

4.4. Interval scale

The interval scale gives us a great deal of more possibilities for analysis. This scale represents measures that can be added together. As an example, the size of the compiled code is in interval scale. The compiler adds some bytes to an executable depending on used libraries etc. Even a simple "Hello World"

program can be several kilobytes in size although it can be coded in a couple of lines. It's not reasonable to say that this executable is twice as larger as the other one in bytes. We can reasonably say that this executable is larger that the other one. And most importantly we can add 200 bytes to measure of 2000 or 200000 bytes and the result reflects increase in size.

4.5. Ratio scale

The ratio scale gives us a measure with meaning of twice as larger (multiplication). Ratio scale includes also value zero. An example is lines of code. It is meaningful to say that a piece of code is twice as large as another one when measured in lines of code (or function points)

4.6. Absolute scale

The absolute scale represents items that are counted. Two separate measures of absolute scale give equal values. Scale is absolute in the sense that there is only one possible value. Examples are the amount of faults or the size of staff.

4.7. Measuring code size

Usually code size is measured in Lines of Code (LOC or KLOC - kilo LOC) or Function Points (FP). LOC is very easy to count; almost all editors show line count. Problem is, which lines to count or are lines equal. Usually comments are dropped and only source lines are counted (source LOC, SLOC or KSLOC). In some cases this is enough, but there are languages like C where there are macros and include files which make the concept of a source line a bit stretched. It's possible to calculate lines after pre-processing but such files are rarely available for counting. LOC is also dependent of coding style. Especially with C, it's possible to put many statements on one line. To get comparable LOC counts, authors have to adhere to same standard of coding. [Rosenberg, 1997].

Function point analysis (FPA) attempts to count the size of product based on functionality of the product. Function point count is subjective; there is no guarantee that two persons would get same function point count out of same product. It doesn't mean that FPA is useless, but it has some inherent

inaccuracy. Greatest advantage of FP count over SLOC is that FPs can be counted already at design phase. SLOC count is estimation until product is ready.

4.8. Measure and indicator

An indicator is a presentation of measures. A set of measures can be presented in a great many ways. The presentation affects the usefulness or expressiveness of measures. It doesn't mean that there is one right presentation for a set of measures. It just means that some presentations are better than others are. Here's an example that should clarify things.

Consider the example in the Figure 4.1. There, the number of faults and the lines of code have been measured from eight functions. Functions 1, 2 and 4 are private and the rest are public. Function is *entity*, the target of the measurement. Number of faults, source lines of code (from now on SLOC) and modifier are *attributes* of the entity. Measures are values of the measured attributes.

Scale of the modifier is nominal, you cannot add two private together to get a public or vice versa. Scale of faults is absolute. There clearly is a zero value and it’s meaningful to say that function6 has three times more faults than function4. Scale of SLOC is a bit more problematic. It may seem that scale would be ratio, but function with size 0 exists only as an idea. Actually, it depends on calculation method and coding standard if there even is a value 1. For example a function written in C

```
int power(int x) {return x*x}
```

Or written with only one statement per line

```
int power(int x) {
        return x*x
}
```

In the latter form, SLOC would be two or three. So if the coding standard demands that there should not be two or more statements on one line smallest possible SLOC count for a C function is 2 (or three if we count the ending bracket as a line). Scale type is ratio, weather value zero really exists or not.

	Faults	SLOC	Modifier
Function1	12	34	Private
Function2	3	23	Private
Function3	14	56	Public
Function4	2	45	Private
Function5	9	31	Public
Function6	6	28	Public
Function7	7	41	Public
Function8	15	39	Public

Figure 4.1 Three measures from eight functions.

Presenting just the pure measures is quite uninformative and hard to read. It is also hard to say what these base measures actually tell. Figure 4.2 presents measures SLOC and the number of faults as a chart. Each dot presents one function. There seems to be a trend; the smaller functions have fewer faults than larger ones. That's not a surprise, but you would expect that even these simple base measures could tell a bit more.

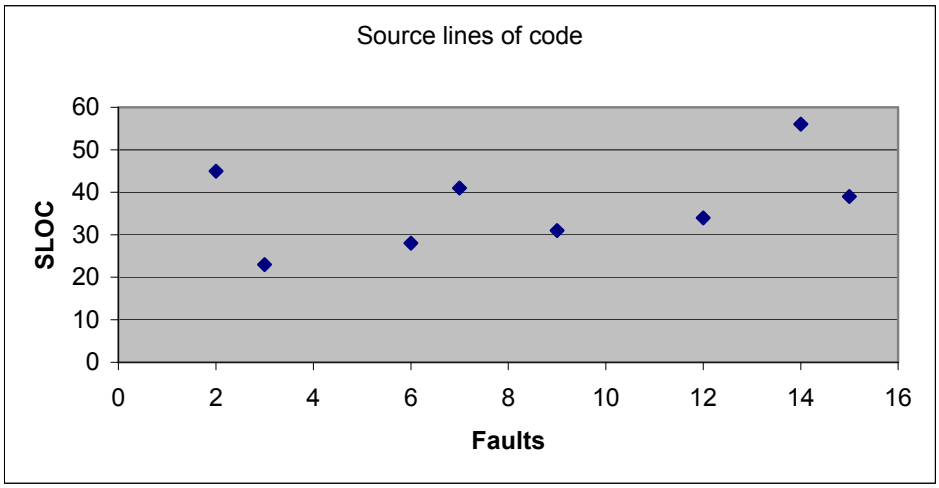

Figure 4.2 Measures from example set of data in a scatter plot chart

Figure 4.3 shows a *derived measure* faults/SLOC in table. A derived measure is calculated from two (or more) base measures. In this case, we simply divide number of faults with source lines of code for each function. Derived measure may be a result from much more complicated calculations. It may also include aggregation of data.

	Faults/SLOC
Function1	0,352941176
Function2	0,130434783
Function3	0,25
Function4	0,044444444
Function5	0,290322581
Function6	0,214285714
Function7	0,170731707
Function8	0,384615385

Figure 4.3 A derived measure faults/SLOC from example set of data

Figure 4.4 presents the derived measure faults/SLOC with a threshold in a chart. In this case, threshold presents a previously set quality limit. If faults/SLOC derived measure is bigger than threshold quality limit is exceeded.

A threshold gives a context for understanding the derived measure faults/SLOC. In this case the meaning is very simple; all functions that have lower faults/SLOC are good and those that have higher are bad ones. What is actually done to the "bad" functions depends on what the threshold was meant for. It could be limit for reinspection or rewriting the function.

In this example, the indicator (Figure 4.4) was based on derived measure and a threshold value. Base measures can be used in indicators, for example McCabe's cyclomatic complexity [McCabe, 1976] is a base measure that has been used to indicate the need of rewriting a function.

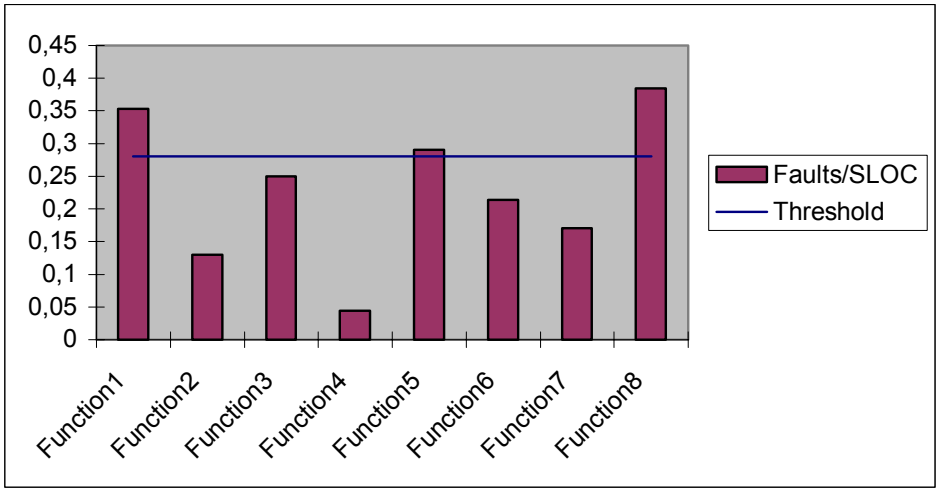

Figure 4.4 Derived measure faults/SLOC with threshold.

Figure 4.5 shows the relationship of measures and an indicator. Measurement of an attribute gives us a *base measure*. A *derived measure* is result of a function of two or more base measures. A *measure* is either a base measure or a derived measure. An *indicator* is a presentation of one or more measures with a context to understand the meaning of measures.

4.9. Defining Measures and Indicators

Measures and indicators are useful only if they fulfil a need of information. There has to be a question or a problem for which we need more information. The problem and our need of information are directly related to certain entities of the real world. In addition, the problem in a sense leads us to suitable measures. We have to select measures that can give an answer to our problem. Other information may be useful, but it doesn't help us to solve this particular problem. When we decide on a useful presentation, again we need to keep the problem in mind. An indicator has to tell something useful to help us to solve the problem. Presentation and interpretation are tied to the problem at hand. So a problem or a need of information leads us to meaningful measures, indicators and interpretation.

An indicator presents data for the information need with decision criteria that helps to understand the data. Understanding is based on the information need we have and decision criteria that helps us to relate the meaning of the measures. If we want to know why failure correction takes so long, we need suitable measures for proper assessment of correction time. The problem guides us on defining the measures. The problem also helps us to choose the right way to present the data. If an indicator helps us to see why failure correction takes time, it's probably a good indicator.

Measurement Information model [Card, 2001 and ISO/IEC CD 15939, 2000] (see Figure 4.5) ties the *information need* and measures together. The information product is a "combination of indicators and interpretations" [Card, 2001, pp 49]. It is an answer to the information need.

A measurable concept identifies the targets of the information need and measurement. The information need may involve more than one entity. The attributes of the entity are mapped to measures. Measures are used to get indicators and indicators are used in the information product.

Path from measures to indicators was covered in Chapter 4.8. An information need is the basis for defining measured entity and measures. An information need also guides the selection of an indicator. The indicator has to provide useful information for the information need. A good indicator tells

when there may be need for action. It supports decision making by highlighting possible problems.

The information product contains indicators with relevant data for interpretation of indicators. It is unwise to base decision just on the indicator. You have to consider the relevance and reliability of the data in the indicator. Data may be from a similar project, but in some aspects, the other project differs so that the indicator is not applicable. As an other example, consider an indicator presented in Figure 4.4. Functions one, five and eight are over the threshold. What if we knew that function one was not coded according to the standard of the organisation? The code is tighter than the norm, i.e. there are fewer lines than by the standard. It would mean that if the code was made by standard it may well have been within the threshold. And no other action than adherence to standard would be needed.

4.10. Indicator and interpretation

An indicator presents a set of data combined with decision criteria. Decision criteria give us something to compare measures to. For final decision making we also need to know what extent we rely on the data. We may have only partial data, data quality may have problems or there may be limitations from data analysis. All the aspects that affect to decision-making are collected with indicators to an information product. Information product gives answer to the information need. Final interpretation is based on all the aspects presented in information product

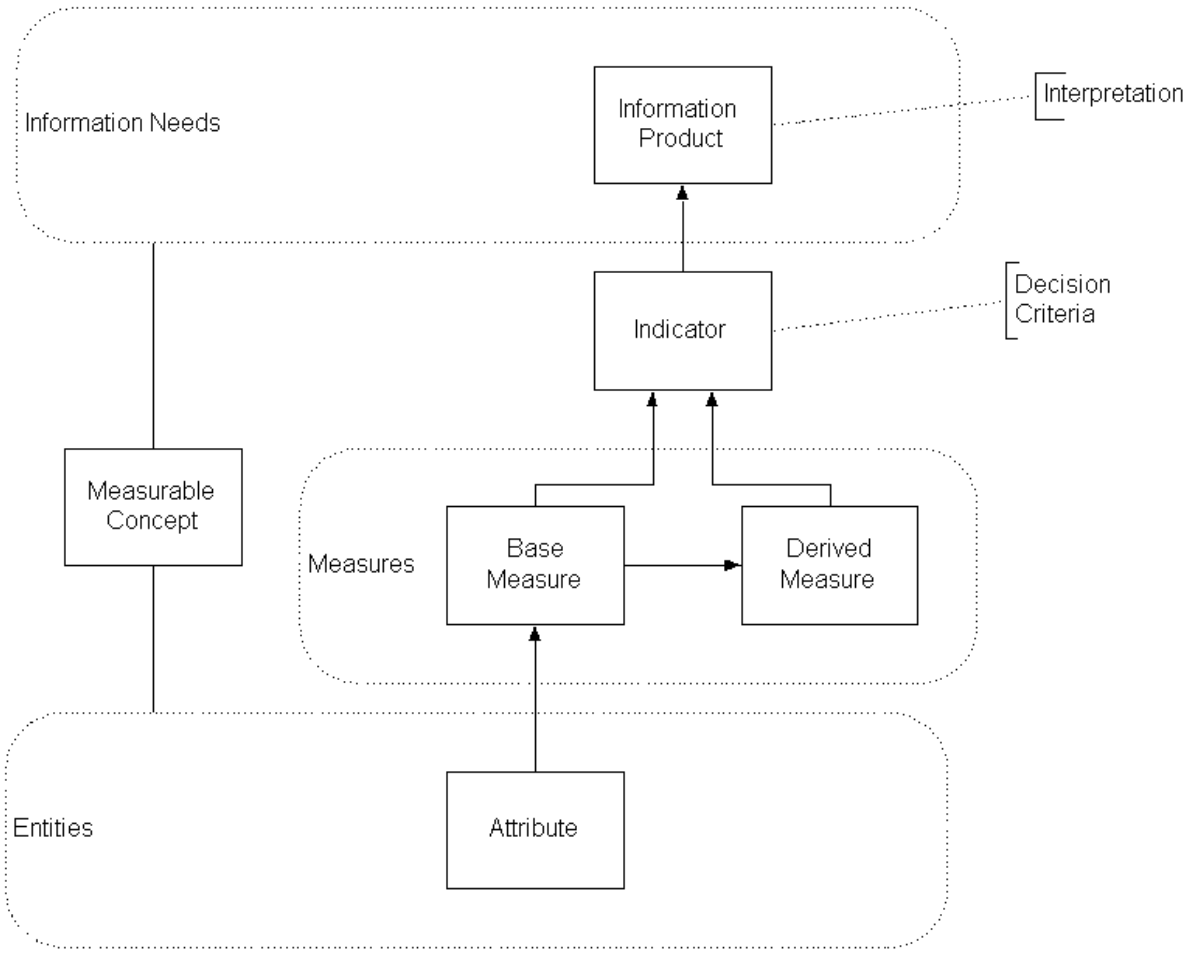

Figure 4.5 Measurement Information Model

In information product, we combine indicator with all the aspects that affect on how we should interpret the data. The interpretation should give us insight and understanding on what is happening in the real world. An information product is based on analysis of measures. The measures reflect the real world and an indicator should reflect on what is happening in the real world.

4.11. Usage of the software measurement

Measures and indicators are based on an information need. There are three basic categories of information needs, three ways to use software measurement: prediction, assessment and process improvement.

Usually prediction comes first. How long does it take to finish this project? How many engineers do we need for it? How much does it cost and so on? Prediction is based on historical measures of previous projects (and their

processes and work products). Planning is based on estimations or predictions. Prediction is based on historical data. Estimation is based on guesses or experience. This does not mean that estimations are inferior to predictions. In some cases estimations give much better results than predictions based on historical data. A person with a long experience may be able to give excellent estimations.

At planning we make estimates how project will come out. As the project is proceeding, we want to know how we are doing. Are we still in the schedule and the budget? We compare plans to measures of the current situation. Have all the activities that we promised to do this far been finished? How does the effort prediction relate to accomplishments and so on? It is important to notice that if a plan doesn't contain measurable goals, it will be very hard to compare any measures against plans.

In a project we may run into process related problems. Indicators may show problems at processes or there may be other needs to improve or speed up processes. Measures can be used to identify improvement areas, for instance measuring code inspections may show that fault fixing and follow-up takes most of the calendar time of the inspection. Process improvements should always be evaluated. It is senseless to implement a change to process that does not improve it. The only way for us to really know that we are actually improving process is to measure that the new process is better than the old one. The information need could be "Is the new process more cost effective than the old one?"

All three types of measurement uses are tied to each other. Measures that were used for assessment will become a part of historical data and can be used in the future for prediction. Predictions themselves are measures and can be used to improve predictions in the future. Measures used for assessment may indicate needs for process improvement. Measures are related to information needs, but the same measures can be used for several different information needs. On the other hand, it is sometimes hard to categorise measurement uses (prediction, assessment or product improvement) for instance when we compare the current situation to our plan and come to the conclusion that we

need to re-plan. Current situation is then used in assessment and also at re-planning for a new prediction.

Prediction is based on historical data, but where does historical data come from? It may come from a measurement database or documatation of old projects. Measures are not used once and then discarded. They should be stored for later use. We need historical data for prediction and improvement evaluation. Historical data can be used at assessment to calculate trends or cumulative counts for example total number of faults. The threshold we used at Figure 4.4 is probably based on historical data.

If we are going to reuse old measures they have to be collected consistently using always a method that gives us comparable measures. Collection method may change, but it has to change so that measures will stay comparable. For example, a size measure is based on source lines of code and it is collected by hand. The measure is comparable if exactly same rules are always followed. Collection method may change to semiautomatic, where counting is done with a program that follows the same rules as before. The data collection may be still more automated by linking the SLOC counting program to a configuration management system, so that each time a file is updates to the storage system, SLOC is counted.

Historical data gives us a reference for comparison. A measure tells very little if we cannot compare or relate it to something. A SLOC count of 1000 may seem a high value, but compared to pieces of code that has average SLOC 12000 it is small indeed. Decision criteria in indicators are very often based to historical data.

5 Roles

In this chapter we go through several roles that each offer a unique view towards code inspection. In the next chapter, we define information needs and measures for each role. Each role has a unique view to code inspection, but information needs or measures may be similar.

All roles are a bit exaggerated and caricaturised to highlight each roles unique view towards code inspections and its effects on software development. One person may have several roles or a role may be divided to several persons. Roles are based on distinct views and activities so it should be relatively easy to identify representatives in other organisations. Similar roles can be found from software engineering literature [Pfleeger, 1999].

This list of roles is not complete; a person that does software measurement work is not represented. There is little sense to introduce a role, whose work we are at least partly describing.

This work covers only roles that are related to code inspections, to keep the number of roles manageable. If other inspection types were also included, we would need roles like requirement analyst, software architect, software designer etc.

Roles are based on a simplified waterfall lifecycle (see Figure 3.1) and on activities around code inspection.

5.1 Software Engineer

Before code inspection can be arranged there has to be the code to be inspected. A software engineer's task is to create code that fulfils design. The implementation is based on previously made and accepted design and requirement documents. The software Engineer's focus is on creating good and solid code following standards and guidelines of the organisation.

In inspection, a software engineer has both author's and checker's roles. Fellow software engineers act as checkers and possibly as a secretary or a chairman.

For author, the most important outcome of inspection is a list of faults. The number of faults reflects on quality of code. Inspection is also feedback to developer. Faults should be seen as learning opportunity. People do make mistakes, but we also learn from mistakes. The author is not the only person who can benefit from the inspection, all participants can learn from it.

5.2 Test Engineer

The test engineer is responsible of software testing. The testing starts after code inspection (usually). Inspection is probably the most efficient way to remove

faults from software product, but it is unreasonable to use only inspection to detect faults. Testing should compliment inspections and other problem finding activities. They all are essential parts of building quality software. Test engineer's task is not just to find failures, but also to validate that the software works as was promised in requirements.

For a test engineer inspection tells something about the quality before he starts testing. If the quality is already good testing may be relaxed, but if the quality is poor, then perhaps it is time to run some extra tests.

5.3 Quality engineer

The quality engineer (QE) is responsible of quality of products and processes. There's not much she can do directly to product quality, but she can have an effect to the quality of processes. An efficient inspection process leaves fewer faults in code, so effectiveness of inspection is one of a QE's concerns.

A checking rate guides the efficiency of the inspection. A QE has to promote proper checking rate in the project and follow that it is applied in inspections. A QE should have experience on inspections, so she can guide the inspection process. She should also suggest reinspection if it seems that a poor inspection has compromised the quality of the end product.

5.4 Process Engineer

Both the whole software development process and single processes concern process engineer (PE). They provide guidelines and adjust checking rates to get most out of inspections. Where a QE works with single inspection or inspections of one project, a PE looks after all inspections in the organisation. A QE's scope is in single cases and a PE's in whole system. Their interest on inspection is very similar, but scope is different.

The inspection practice has to fit in the overall development process and it has to run smoothly and efficiently. PEs also pilot, evaluate and introduce changes to processes.

5.5 Project Manager

A project manager's task is to keep the project in control, i.e. in the schedule and within the budget. She is interested on how much time inspections take (effort), how long the inspection lifecycle is and of course are inspections efficient. Efficient inspections (i.e. time is not wasted and a good number of faults are found) is a good way to keep a software project under control. Large variations from norm should be examined and possibly reinspected. Gilb and Graham [1995] use derived measure faults remaining (Gilb and Graham call it defects remaining) as an exit criteria for inspection. Percentages of faults found

(effectiveness) can be calculated, if the inspection process is stable. Effectiveness and the number of faults found in the inspection tell how many faults remain unfound in the code.

6. Information needs and measures of code inspection by generic role

In this chapter, we will define one information need for each role. Our list of information needs is not exhaustive. It is more like a set of examples of information needs and what measures and indicators can be derived from them. Emphasis is on measurement information model, information needs, measures and indicators. Measurement information model is used to tie information needs to measures and indicators.

All of the information needs, measures or indicators can be found in some form in literature. Formalisation of these information needs and measures with measurement information model is new. We believe it offers new insight to measurement.

Format for the measurement information model (see Figure 6.1) is the same as in standard ISO/IEC CD 15939 [2000]. We will define a combined measure checking rate in Chapter 6.1 and at the same time we look a bit closer what kind of presentation for measurement information model offers. In Chapters 6.2, 6.3, 6.4, 6.5 and 6.6 we define one information need for each role and in chapter 6.7 we will see how different measures are used in information needs.

6.1 A common measure: Checking rate

Gilb and Graham [1993] use checking rate to maximise inspection efficiency. The highest number of faults can be found using the optimum-checking rate. The optimum checking rate may be different at different organisations, but recommended staring rate should be one (logical) page per hour [Gilb and Graham, 1993]. Using the optimum checking rate should give us a good estimation of the fault detection rate. The fault detection rate is a percentage of faults that are found at a given phase or activity (compared to the total number of faults found in current phases and later). Then from the fault detection rate and the number of faults found we could calculate an estimation of remaining faults.

Checking rate can be used as an entry criterion for inspection meeting: If the checking rate is too high, then the inspection meeting should be postponed and inspectors should prepare better. Efficiency of inspection is related to preparation and inadequate preparation leads to inefficient inspection. Inspection meeting itself is quite time consuming and of course, all development time should be used to efficiently

Information Need	Assess preparation for code inspection
Measurable Concept	Preparation
Relevant Entities	Code inspection (inspection record)
(Measurable) Attributes	1. Size of inspected code 2. Effort used for preparation
Base Measures	1. SLOC of inspected code 2. Time used to checking
Measurement Method	1. Count lines of code 2. Estimate used time/Stopwatch
Type of Method	1. Objective 2. Objective
Scale	1. Integer from one to infinity 2. Integer from zero to infinity
Type of Scale	1. Ratio 2. Ratio
Unit of Measurement	1. Logical page (i.e. 30 SLOC) 2. Hour
Derived Measure	Checking rate
Function	Average of checking time divided by checked size
Indicator	Checking rate should be close to optimum checking rate
Model	
Decision Criteria	

Figure 6.1 Information need: "Assess preparation for code inspection"

Gerald Weinberg [2001] is against required or recommended checking rates. He thinks that the checking time itself tells something about the quality of the

checked document. On the other hand, if the checking rate corresponds with the fault detection rate then we can estimate remaining faults, but we loose information on how easy or hard document was to go through.

We will define here first a combined measure for checking rate. Later we will derive optimum checking rate based on this measure.

Terms written in **bold** in a few next paragraphs refer to keywords in measurement information model template presented in Figure 6.1 (and other figures presenting information needs).

Information need "Assess preparation for code inspection" states that we are interested in how well inspectors have been preparing for the inspection. The amount of time used for checking before the actual meeting effects on how many faults are found. Poor preparation may lead to almost useless code inspection. Then, preparation is a **measurable concept** for code inspection. We assume that all the measures we need are already recorded to inspection record, so there is only one **relevant entity** – the inspection record. The target of measurement is actually the inspection, so the entity in this information need should be the inspection. The inspection record presents an inspection and although we measure the inspection, all information is recorded to an inspection record.

Measurable attribute describes what aspects of the entity we are interested in. Size is an attribute of code (and then code inspection) and individual effort is an attribute of the inspection. **Base measures** describe how an attribute is measured. Size could be measured at least as lines of code and function points. Checking rate is traditionally measured from lines of code. Effort is easiest to measure as an interval, duration of time.

Measurement method describes how we get measures. In this case, numbers are already recorded to an inspection record, but we still state here how those numbers are acquired so that we can assess their reliability. **Type of method** can have only two values: objective and subjective. Size in SLOC is objective (if it is calculated by strict rules) since it can be calculated automatically with program and the result is not dependent of a collector. Effort could be subjective since it depends on inspectors' judgement or

objective if an inspector reserves given time just for checking and uses timing device.

Scale tells the range measure may have. **Type of scale** classifies this particular scale in one of the five scale types used in statistics. Type of scale is important to check so that we use only applicable statistical operations to our measures. **Unit of measurement** tells when we have comparable measures. If we had two measures with unit hour, we could compare them together. Unit of measurement sets or depends on extent of observation. If we are interested in effort of a large project then hour as unit of measurement is awkward. Man-month or even man-year is a more appropriate unit.

Derived measure is a function of two or more base measures. Here we name derived measures used for indicators. **Function** describes algorithm or calculation used to combine base measures. We must keep in mind that scale types may set limitations to usable functions. Our derived measure is checking rate, i.e. how many (logical) pages have inspectors checked per hour on average. Our function describes how checking rate is calculated exactly.

Indicator evaluates derived measures in relation to information need. **Model** contains statistical (or other) analysis made for derived measures for decision-making. **Decision criteria** set limits where derived measures should be. Indicators are presented to measurement users in information product. Information product contains also information about uncertainty, accuracy and importance of indicators. In this case, we skip model and decision criteria. Later chapters contain examples of how derived measures can be presented.

Figures 6.2, 6.3 and 6.4 contain measures from three inspections for information need "Assess preparation for code inspection"; the number of faults found will be used later. Average preparation times are for inspection A 0.9 h (0.5+1.25+1.5+1.25+0.5+0.25+1.25 h /7), inspection B 1.7 h and inspection C 0.5 h. Checking rates are for inspection A 7.5 pages / hour (7 pages/(0.5+1.25+1.5+1.25+0.5+0.25+1.25 h /7)), inspection B 1.8 pages / hour and inspection C 4.0 pages / hour.

Inspection A	Preparation time (in hour)
Inspector A1	0.5
Inspector A2	1.25
Inspector A3	1.5
Inspector A4	1.25
Inspector A5	0.5
Inspector A6	0.25
Inspector A7	1.25
Size of document (in pages)	7

Figure 6.2 Measures of Inspection A for information "Assess preparation for code inspection "

Inspection B	Preparation time (in hour)
Inspector B1	1
Inspector B2	1.5
Inspector B3	2.25
Inspector B4	2
Size of document (in pages)	3

Figure 6.3 Measures of Inspection B for information need "Assess preparation for code inspection "

Inspection C	Preparation time (in hour)
Inspector C1	0.25
Inspector C2	0.25
Inspector C3	0.5
Inspector C4	1
Inspector C5	0.5
Size of document (in pages)	2

Figure 6.4 Measures of Inspection C for information need "Assess preparation for code inspection "

As we can see from inspection measures in Figures 6.2, 6.3 and 6.4, the preparation time is separated from the actual inspectors. There is no sense to record inspectors too. We are interested in how the inspection process works not how individuals accomplish. Besides, giving inspectors anonymity in data collection tends to improve data quality.

6.2 An information need and related measures for a software engineer

A software engineer makes the code to be inspected. She should be able to be proud of her accomplishments. On the other hand, a code inspection detects faults, so she should be able to learn from them. Software engineers information needs could be like "How well did I manage with this piece of code?" and "How could I program better in the future?"

Figure 6.6 shows an example indicator for information need " How well did I manage with this piece of code?" Fault density is compared to the average of author's fault densities. If fault density is below average then the latest piece of code is ok (if inspections detect faults equally). The indicator in Figure 6.6 contains very little information: average and fault densities in the same order as the inspections were held. The indicator in Figure 6.7 shows same data points in different form. Fault density is replaced with a scatter plot of numbers of faults and lines of code. The average value is multiplied by lines of code. The scatter plot presentation gives us some sense how big pieces have been inspected and how size and amounts of faults relate to each other. On the other hand, we have lost the ordering. In a small data set, it would be possible to add ordinal value to each data point to indicate its order, so we wouldn't loose any information that was available previously.

Indicator in Figure 6.7 shows also the potential dangers of badly chosen measures. Consider a situation where salary or bonuses are affected by decrease in fault density. Fault density decreases as quality improves, but dividend of fault density is size as lines of code. It may be possible to decrease fault density by increasing the dividend – size. Fault density may indeed decrease, but the wanted effect; quality improvement will not be achieved. Some derived measures like productivity as lines of code / hour and fault

density as faults /lines of code are subject to manipulation and they should not solely be used as a basis for awards. They should be used in conjunction with some other measures that detect or prohibit manipulation

Information Need	How well did I manage with this piece of code
Measurable Concept	Code quality
Relevant Entities	1. Code inspection record 2. Unit of inspected code
(Measurable) Attributes	1. Faults found in authors code inspection 2. Size of inspected code
Base Measures	1. Faults in inspection of code X 2. Lines of code X
Measurement Method	1. Count (major) faults from code X inspection records 2. Count source lines of code from code X
Type of Method	1. Objective 2. Objective
Scale	1. Integers from zero to infinity 2. Integers from one to infinity
Type of Scale	1. Ratio 2. Ratio
Unit of Measurement	1. Fault 2. Line
Derived Measure	Inspection fault density
Function	Divide faults by lines of code
Indicator	Code fault density compared to authors average fault density
Model	
Decision Criteria	

Figure 6.5 Information need: "How well did I manage with this piece of code?"

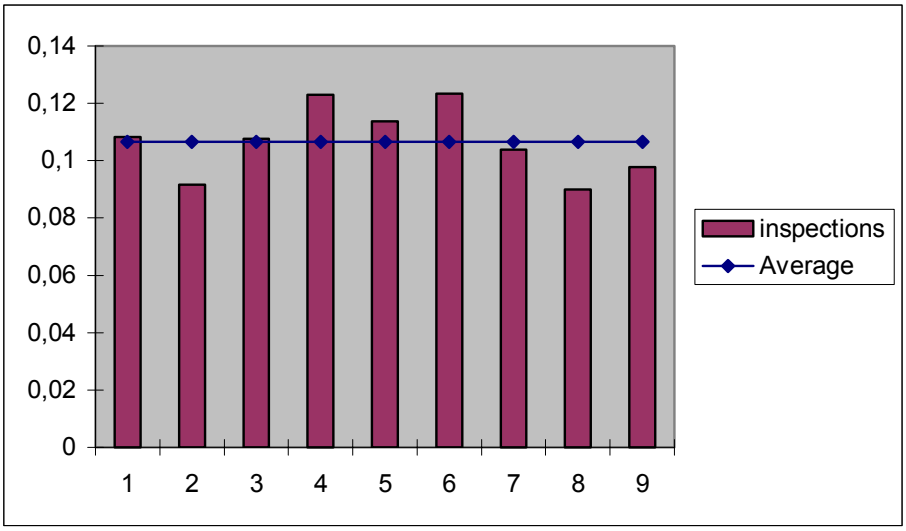

Figure 6.6 Indicator for " How well I managed with this piece of code?"

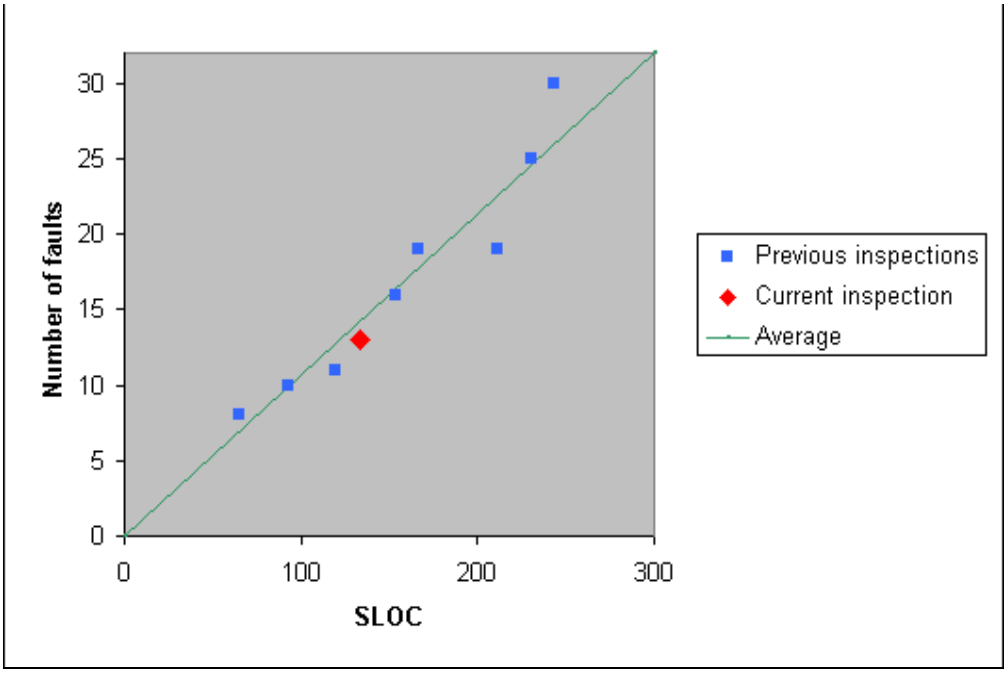

Figure 6.7 Another indicator for " How well did I manage with this piece of code?"

6.3 An information need and related measures for a test engineer

Effectiveness of testing is like effectiveness of inspection; it tells how many percents of failures testing detects. It is reasonable to assume that in large

projects effectiveness of testing is more or less equal. So let's assume that effectiveness of testing is about 50 %. Then from 100 failures testing discovers 50 failures or from 500 failures testing discovers 250 failures. Of course, we would concentrate our testing efforts to the code piece with 500 failures, since improvement in effectiveness gives bigger payoff. If we could also estimate how many faults remain in code, would it then be sensible to test more rigorously the code that probably has more faults and probably causes more failures?

Information Need	Extent of testing
Measurable Concept	Code quality
Relevant Entities	1. Code inspection record
(Measurable) Attributes	1. Faults found in authors code inspection 2. Size of inspected code
Base Measures	1. Faults in inspection of code X 2. Source lines of code
Measurement Method	1. Count (major) faults from code X inspection records 2. Count lines of code
Type of Method	1. Objective 2. Objective
Scale	1. Integers from zero to infinity 2. Real from zero to one
Type of Scale	1. Ratio 2. Absolute
Unit of Measurement	1. Fault 2. SLOC
Derived Measure	Remaining faults per SLOC
Function	Faults in inspection of code X times (100 – effectiveness) /(effectiveness times SLOC)
Indicator	Distribution of remaining faults
Model	
Decision Criteria	

Figure 6.8 Definition of derived measure "Extend of testing"

	# faults	SLOC	faults/page
1	5	31	5
2	7	32	6
3	17	47	11
4	29	53	16
5	7	53	4
6	11	54	6
7	16	59	8
8	20	65	9
9	21	66	10
10	22	71	9
11	30	74	12
12	9	78	4
13	22	79	9
14	29	84	10
15	32	86	11
16	37	107	10
17	44	108	12
18	53	129	12
19	48	134	11
20	53	138	12
21	57	151	11
22	100	167	18
23	56	173	10
24	62	177	11
25	66	177	11
26	101	182	17
27	69	183	11
28	67	191	10
29	76	192	12
30	60	200	9

Figure 6.9 Measures for information need "Extend of testing" from 30 inspections.

Inspections detect faults and testing detects failures. Faults cause failures, but not all faults. Same faults are in parts of code that may never be executed or

sometimes a failure hides other failures. The amount of faults still should reflect to amounts of failures. So if we estimate how many faults remain in inspected code we may find pieces of code that are more likely to contain failures.

Information need "Extent of testing" in Figure 6.8 is based on the assumption that a large number of remaining faults indicate a greater possibility of failures. Figure 6.8 presents a derived measure Remaining Defects, which is based on the effectiveness of inspection (defined in Chapter 3.2). For the definition of remaining defects, we assume that a reliable value of effectiveness of code inspection is available (such a value would probably come from a process engineer).

Figure 6.9 presents a data collection from 30 inspections. Faults and SLOC are base measures and faults per page is a derived measure. The amount of pages is SLOC divided by 30 and derived measure is faults divided by pages.

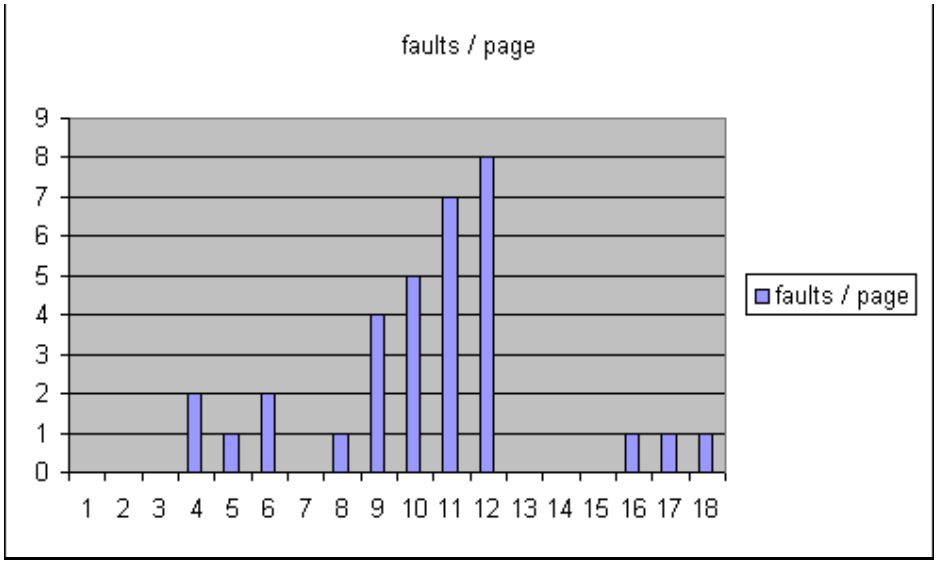

Figure 6.10 Distribution of faults/page in information need “Extent of testing”

Inspection with 16,17 and 18 faults / page in Figure 6.10 just jump out. They are prime candidates for thorough testing. On the other hand, inspections with only four, five or 6 faults per page could go through a bit easier testing. It is

important to remember that such conclusions are only valid if inspection process and effectiveness is stable. It should be also verified statistically that (very) low amount of remaining faults really mean a very low change of finding failures in testing. Statistical analysis and verification could lead to some kind of control limits that could be also used as decision criteria.

6.4 An information need and related measures for a quality engineer

A quality engineer is interested in how an inspection was conducted, was it within acceptable limits or did it greatly deviate from the norm. The inspection process should be stable, so that unusual inspections can be recognised from the noise generated by natural process deviations. One aspect of inspection is preparation.

Information Need	Was preparation to code inspection acceptable?
Measurable Concept	Preparation
Relevant Entities	Code inspection record
(Measurable) Attributes	1. Effort spent on individual checking 2. Size of checked code unit
Base Measures	1. Time used in individual checking by X 2. Lines of code in unit checked by X
Measurement Method	1. Reported by inspector X 2. Count source lines of code from unit checked by X
Type of Method	1. Objective 2. Objective
Scale	1. Real numbers from zero to infinity 2. Integers from one to infinity
Type of Scale	1. Ratio 2. Ratio
Unit of Measurement	1. Hour 2. Line
Derived Measure	Checking rate
Function	Average of time used divided by lines of code per inspector
Indicator	Code checking rate

Model	Compute control limits using statistical process control like Porter [2001]
Decision Criteria	Results outside of control limits are reinspected

Figure 6.11 Information need " Was preparation to code inspection acceptable?"

6.5 An information need and related measures for a process engineer

Some of combined measures already used in previous information needs come from a process engineer. A process engineer works with same subjects as quality engineer, but the scope of process engineer's work is much wider. Where quality engineer works with measures from one inspection or any subset of all the inspections in one project, a process engineer may work on inspections from whole organisation. Optimum checking rate is one example of such measures.

Information Need	What is the optimum checking rate for code inspection?
Measurable Concept	Preparation
Relevant Entities	Code inspection record
(Measurable) Attributes	1. Size of inspected code 2. Faults found in inspection 3. Effort used for preparation
Base Measures	1. SLOC of inspected code 2. Number of faults found at inspection 3. Time used to checking
Measurement Method	1. Count lines of code 2. Count found (major) faults 3. Ask inspector for time used in checking
Type of Method	1. Objective 2. Objective 3. Subjective
Scale	1. Integer from one to infinity 2. Integer from zero to infinity

3. Real from zero to infinity	
Type of Scale	1. Ratio 2. Absolute 3. Ratio
Unit of Measurement	1. Logical page (i.e. 30 SLOC) 2. Faults 3. Hour
Derived Measure	Optimum checking rate
Function	Average of checking time divided by size in inspection X
Indicator	Average checking rate compared to number of faults
Model	
Decision Criteria	

Figure 6.12 Information need " What is the optimum checking rate for code inspection?"

6.6 An Information need and measures for a project manager

A project manager needs to know if her project is in budget and schedule. Software inspections take their time and it has to be taken into account in planning. When important deadlines approach, project manager probably wants to know if all code inspections can be finished in time.

Information Need	Can we finish all code inspections in time?
Measurable Concept	status of code inspections
Relevant Entities	1. Project plan 3. Inspection records 4. Calendar
(Measurable) Attributes	1. Deadline of code inspections 2. Status of inspection X 3. Today
(**Base) Measures**	1. Last day of the coding phase 2. Status of code unit inspection 3. Date of today
Measurement Method	1. Collect date from plan 2. If there is no record, value is "not started", otherwise if

	meeting has not been held value is “started”, if inspection is not yet finished value is “meeting held”. For finished inspection value is “finished” 3. Check from calendar
Type of Method	1. Objective 2. Objective 3. Objective
Scale	1. Date 2. “not started”, “started”, “meeting held” or “finished” 3. Date
Type of Scale	1. Interval 2. Ordinal 3. Interval
Unit of Measurement	1. Day 2. 3. Day
Derived Measure	
Function	
Indicator	
Model	
Decision Criteria	

Figure 6.13 Information need “ Can we finish all code inspections in time?”

As the indicator in Figure 6.14 shows, there are three unfinished inspections left. Two of them, inspections 10412 and 55, are late and inspection 10412 is in danger not to even finish until deadline. There are at least two options available: delay deadline or push inspection 10412. Delaying deadline because one inspection may be late seems a bit too drastic. Pushing inspection should be possible since there are still 16 calendar days left and normally inspection takes 18 days of calendar time. To be on the safe side, project manager could push all three unfinished inspections.

Duration from "not started"	18
Duration from "started"	10
Duration from "meeting held"	4
Today	15-Jan-01

ID	Status	Target Date	End of phase
10412	not started	*22-Jan-01*	*31-Jan-00*
55	Started	*21-Jan-01*	31-Jan-00
A3	Meeting held	25-Jan-01	31-Jan-00
SZ33	Finished	07-Jan-01	31-Jan-00

Figure 6.14 Example indicator for information need "Can we finish all code inspections in time?" Alarming figures would be in red and good figures in green if colours were used. Here alarming figures are in italics and good figures are in grey.

Indicator uses traffic lights as a metaphor. Red means that things are wrong and something should be done or it may even be too late to do anything within current schedule. Green means that things are in good shape and its ok to proceed. Yellow is a sort of warning sign. Things may still be ok, but they are slipping. With a bit more effort, things would probably be in schedule again. Interpretation of traffic lights depends on the usage, but it is an effective way to tell where to concentrate efforts.

The indicator in Figure 6.14 has a weakness that is not apparent. The indicator does not tell anything about availability of inspectors. It could be that both late inspections (10412 and 55) share one or more inspectors. It could be that an assigned inspector in both late inspections is on vacation or on a business trip abroad. It is possible to change inspectors but what if the author is not available?

6.7 Combined measures

Information needs in previous chapters had several common measures. That's not surprising since all information needs are related to code inspection. It still shows that there are some common measures for inspections (similar measures

can be found in Barnard and Price [1994], Gilb and Graham [1993], Grady and Van Slack [1994], Strauss and Ebenau [1993] and Weller [1993]). In fact, we assumed that most of those measures were already stored to inspection record. Most of those measures are also built into the inspection process. Preparation rate can be used as a criterion whether inspection meeting should be postponed or continued as planned.

In a sense, we assume here that the inspection process is stable and measures are recorded to inspection record. If measures in records are unreliable (due to unstable process for instance) then we have to consider how it effects to indicators.

	size of inspected code	effort used for preparation	faults found in inspection	effectiveness	deadline	status of planned inspection	current date
Assess preparation for code inspection	*	*					
How well did I manage with this piece of code	*		*				
Extent of testing	*		*	*			
Was preparation to code inspection acceptable?	*	*					
What is optimum checking rate for code inspection?	*	*	*				
Can we finish all code inspections in time?					*	*	*

Figure 6.15 Measures and information needs combined

As you can see from Figure 6.15 several measures can be used in different information needs. It is quite natural since there are commonly used measures (size, effort and number of faults). It's good to notice that software engineers information needs can be fulfilled with measures that would be collected anyway to guide the inspection process.

6.8 Decision Criterion

We have seen five roles related information needs and one based on common measure checking rate. Some of the information needs were closely tied to

inspection process (“Was preparation to code inspection acceptable? And “What is optimum checking rate for code inspection?”). Process related information needs can have a distinct decision criteria. Processes used in large scale probably generate enough data for statistical analysis. A Information need can be then related to controlling the process and decision criteria will tell when process is in control or within acceptable limits. For an individual information need like software engineers “How well did I manage with this piece of code” it doesn’t really make sense to have a strict decision criterion. After all, what kind of decision a software engineer would make if there would be significantly more faults than before? The information need is not directed to control development process, just to give software engineer some sense when she has done good work.

7. Conclusions

Generic roles were defined so that they should cover all software inspection related activities. Generic roles also cover several information needs around code inspection. We also wanted to use common measures in information needs. Consequentially information needs, measures or indicators do not offer anything new. Would more specific information needs have led to more precise and specific measures, is an interesting question. Probably generic roles were not a good way to study measurement uses for individuals.

Measurement information model is a quite strict and formal way to describe measurement. It also forces to consider many important (and sometimes neglected) aspects of measurement, like type of scales and what can be done with such measures. Thorough description of measurement may help the validation of measures. Information needs set what measures should tell. Clear documentation on how measures are acquired and handled can then help on validation.

After all, we wanted to find uses for those common measures, but from different angles, in this case from different roles. In many cases the developer is forgotten as a measurement user. Phrases like "fact based management" create a picture of managers using measures to manage. The developer is forgotten, since measures usually tell how she has performed and actions from management direct how she should act in the future. Developer is quite capable to use the measures for micromanagement, i.e. managing her own work. Quite often she could tell how long an action would take, but schedule pressures and inherent untrust prohibits using self-management as source of scheduling.

All information needs in this work don't have models or precise decision criteria. Those information needs are not related to management or decision making, so it is quite natural that setting precise decision criteria does not make sense. Measures and indicators give guidance or suggestions on what could be a good way to proceed. Each of these indicators could be used with precise decision criteria, but would they loose their current connection to non-managing roles. Using for example "Extent of testing" information need with

decision criterion takes some choice away from test engineer. Without decision criteria test engineer has more freedom on how to interpret and use the information.

8 References

[Barnard and Price, 1994] Barnard, J. and Price, A., Managing code inspection information. *IEEE Software*, **11(2)** (March 1994), 59-69.

[Boehm, 1981] Boehm, Barry W., *Software Engineering Economics*. Prentice-Hall, 1981.

[Card, 2001] Card, David, A Practical Framework for Software Measurement. Proceedings of International Conference on Software Management and Applications of Software Measurement, 2001, 43-52.

[Fagan, 1986] Fagan, M. E., Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, **15(3)**, 182-210.

[Fenton and Pfleeger, 1997] Fenton, Norman and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous & Practical Approach,* second edition. PWS Publishing Company 1997.

[Gilb and Graham, 1993] Gilb, Tom and Graham, Dorothy, *Software Inspections*. Addison-Wesley, 1993.

[Grady and Van Slack, 1994] Grady R.B. and Van Slack, T., Key lessons in achieving widespread inspection use. *IEEE Software*, **11(4)** (July 1994), 46-57.

[Humphrey, 1997] Humphrey, Watts S., *Introduction to the Personal Software Process*. Addison-Wesley 1997.

[ISO/IEC CD 15939, 2000] ISO/IEC CD 15939, CD 15939: Software Engineering – Software Measurement Process Framework. 2000

[Land, Sauer and Jeffery, 1997] Land, Lesley Pek Wee, Saure, Chris and Jeffery, Ross, Validating the Defect Detection Performance Advantage of Group Designs for Software Reviews: Report of a Laboratory Experiment Using Program Code. *Proceedings of Australian Software Engineering Conference,* 1997, 17-26

[McCabe, 1976] McCabe, Thomas A., A Software Complexity Measure. *IEEE Transactions on Software Engineering*, **2(4)** (Dec 1976), 308-320.

[Pfleeger, 1988] Pfleeger, Shari Lawrence, *Software Engineering: Theory and Practice*. Prentice Hall, 1988.

[Pressman, 2000] Pressman, Roger s., Software Engineering: A Practitioner's Approach – European Edition. McGraw-Hill, 2000.

[Porter and Votta, 1997] Porter, Adam and Votta, Lawrence, What Makes Inspections Work? *IEEE Software*, **14(6)** (Nov-Dec 1997), pp 99-102.

[Rosenberg, 1997] Rosenberg, Jarrett, Some Misconceptions about Lines of Code, *Proceedings of Fourth International Software Metrics Symposium*. IEEE, 1997, pp 137 - 142

[Strauss and Ebenau, 1993] Strauss, Susan H. and Ebenau, Robert G.: *Software Inspection Process*. McGraw-Hill, 1993.

[Weinberg, 2001] Weinberg, Gerald M., How Much Prep Time Do You Need for Technical Reviews? *The Software Testing & Quality Engineering Magazine*, **3(3)** (May/June 2001), pp 79-80.

[Weller, 1993] Weller, E.F., Lessons from three years of inspection data. *IEEE Software,* **10(5)** (September 1993), 38-45.

[Zuse, 1998] Zuse, Horst, *A Framework of Software Measurement*. Walter de Gruyter 1998.