Software Quality

- Software quality is an abstract concept. Its presence can be difficult to define, but its absence can be easy to see instantly.
- **What is quality?**
- **Views of quality.**

What is Quality?

- A principal objective of software engineering is to improve the quality of software products. But quality, like beauty is very much in the eye of the beholder.

A quality software product

- Users: Works and produces correct results, is available when needed.
- Testers: Compliance to requirements
- Customer: Is not too expensive.
- Marketing Manager: Looks good, has more features than the competitors product.
- Developers: Can be modified and enhanced to user needs.
- **Quality is different things to different people. All are valid but they may not coincide.**

Example

We have €100,000 to improve the quality of a product. How should it be divided between the following factors ?

- Ease of use.
- Performance (response time).
- Reliability .
- Functionality.
- Timeliness.
- …

Example

How does the answer change depending on the product ?

- An ATM.
- A language teaching software for children.
- A CASE tool.
- A computer game.
- A nuclear power station control software.

Definition

Software quality refers to two related but distinct concepts:

- Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software. It is the degree to which the correct software was produced.
- Software structural quality refers to how it meets non functional requirements that support the delivery of the functional requirements, such as robustness or maintainability. It has a lot more to do with the degree to which the software works as needed.

Problems With This Definition

- Some quality requirements are difficult to specify in an unambiguous way.
- Software specifications are usually incomplete and often inconsistent. A software product may conform to its specification, but users may not consider it to be of high-quality.
- Conflict between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusablity, etc.).

Software Quality Attributes

The non functional requirements.

Satisfying non-functional requirements first requires specifying and defining them.

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Quality Conflicts

- It is not possible for any system to be optimised for all of these attributes – for example, improving robustness may lead to loss of performance.
- The quality plan should therefore define the most important quality attributes for the software that is being developed.
- The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as reliability or usability, is present in the product.

Views of Software Quality

- The transcendental view sees quality as something that can be recognised but not defined.
- The user view sees quality as fitness for purpose.
- The manufacturing view sees quality as conformance to specification.
- The product view sees quality as tied to inherent characteristics of the product.
- The value-based view sees quality as dependent on the amount a customer is willing to pay for it.

Transcendental view

- Concerns innate excellence.
- The type of quality assessment we apply to novels, how do you express the qualities of a good book ? The practised reader develops a “good feeling” for this type of quality.
- So too with software the striven-for "recognition" is the transcendental definition of quality.

User view

- Concerns ‘fitness for use’.
- Whereas the transcendental view is ethereal, the user view is more concrete, grounded in product characteristics that meet the user's needs.
- This view of quality evaluates the product in a task context and can thus be a highly personalised view.
  - Reliability.
  - Performance.
  - Usability.

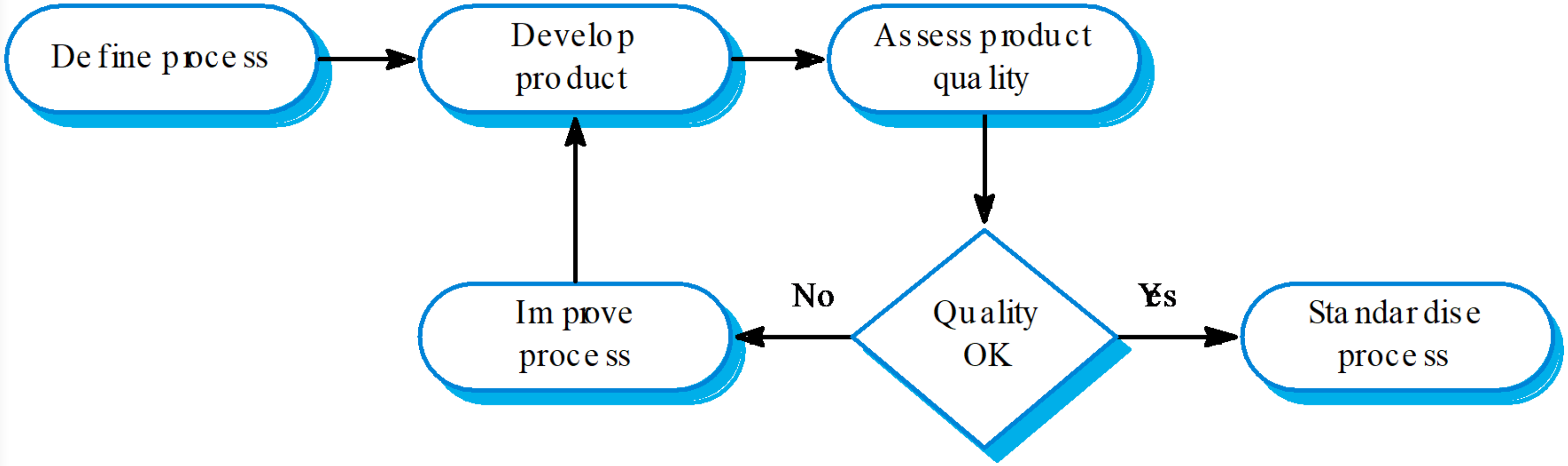
Manufacturing View

- The Manufacturing view focuses on product quality during production and after delivery.
  - Examines whether or not the product was constructed "right the first time," in an effort to avoid the costs associated with rework during development and after delivery.
  - This process focus can lead to quality assessment that is virtually independent of the product itself.

Process and Product Quality

- The quality of a developed product is influenced by the quality of the production process. A good process is usually required to produce a good product.
- For manufactured goods, process is the principal quality determinant.
- Process quality is particularly important in software development as some product quality attributes are hard to assess e.g. maintainability – need to use the software for a period of time.
  - For design-based activity, other factors are also involved The application of individual skills and experience is particularly important in software development;
  - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

Process-based Quality



Product View

- Whereas the user and manufacturing views examine the product from without, a product view of quality looks inside, considering the product's inherent characteristics. Relates to attributes of the software.
- This approach is frequently adopted by software-metrics advocates, who assume that measuring and controlling internal product attributes will result in improved external product behaviour (quality in use).

Internal VS External Attributes

- Internal Attributes
  - These can be measured by examining the object on its own, separate from its behaviour.
- External Attributes
  - These can be measured only with respect to how the object relates to its environment.
- External attributes are more difficult to measure than internal and they are measured later in development.
- A goal of measurement is to identify the relationships among internal and external attributes.

What we measure

- Products – any artefacts, deliverables or documents that result from a process activity.
- Processes – the collection of software-related activities.
- Resources – entities required by the software process.

Components of measurement

Entities	Attributes	
Products	**Internal**	**External**
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity …	reliability, usability, maintainability …
Processes		
Writing specification	time, effort, number of requirements changes …	cost, stability …
Resources		
Personnel	age, price…	productivity, experience …

Product View

- If the measure of an internal attribute is to be a useful predictor of an external software characteristic three conditions must hold:
  - The internal attribute must be measured accurately.
  - A relationship must exist between what we can measure and the external behavioural attribute in which we are interested.
  - This relationship is understood, has been validated and can be expresses in terms of a formula or model.

Value-based view

- Deals with costs and profits. It concerns balancing time and cost against profit.
- Equating quality to what the customer is willing to pay for encourages trade-offs between cost and quality.

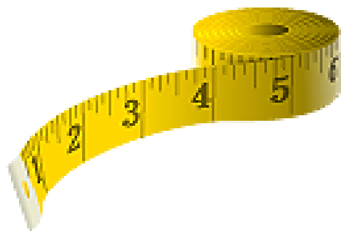
Agreement Is Needed

- Different views can be held by different groups involved in software development.
  - Customers or marketing groups typically have a user view, researchers a product view.
  - The production department a manufacturing view.
  - Software developers concentrate on the product and manufacturing view.
- If the difference in viewpoints is not made explicit, misunderstandings about quality created during project initiation are likely to resurface as (potentially) major problems during product acceptance.

What is quality?

software

+



measures

A good definition of quality must let us measure quality in a meaningful way.

Components of quality

- From a measurement perspective we must be able to define quality in terms of specific software product attributes of interest to the user.
- That is we want to be able to measure the extent to which these attributes are present in our software products.
- We can then set targets for quality attributes in a measurable form

What are Software Metrics?

- metric: a quantitative measure of the extent or degree to which software possesses and exhibits a certain characteristic, quality, property or attribute – software size
- unit of measure: a reference quantity in multiples or fractions in which any quantity of the same kind can be measured - KDSI;
- measurement: a number with an associated unit of measure which describes some aspect of software – 10,000 KDSI

The Need For Metrics

- **Visibility:** Software is essentially abstract and visibility into its structure and status is difficult to achieve. Metrics can help to achieve greater visibility and hence more informed and effective management.
- **Planning and control:** To plan, we need measurements of previous projects similar to the current project of interest. To control, we need measurements of work completed and work in progress, so that a comparison of actual and planned progress can be made and an appropriate action taken when actual falls behind plan.

The Need For Metrics

- **Quality:** This includes reliability, maintainability, usability, and so on. It is widely believed that improving the quality of all software development products can also improve productivity by reducing subsequent rework.
- **Productivity:** There is a natural concern on the part of management to get better value for money in software, that is, to improve productivity within acceptable quality limits.

The Importance of Measurement

- Measurement is important for three basic activities :
  - Measures help to understand what is happening during development and maintenance. We can assess the current situation and set goals for the future.
  - Measurements allow control over what is happening on projects.
  - Measurements encourage the improvement of processes and products.

Internal Product Attributes

- Internal product attributes describe software products in a way that is dependent only on the product itself.
  - Functionality – the functions supplied by the product to the user.
  - Length – physical size.
  - Complexity – of the underlying problem the software is solving.

Functionality - Function Points

- Use a measure of the functionality delivered by the application as a normalisation value.
- Proposed by Albrecht, who suggested a measure called the function point.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessment of software complexity.

Function Point Analysis

- Based on a combination of system function categories. Estimate the number of the following:
  - external inputs and outputs
  - user interactions
  - external interfaces
  - files used by the system.
- A weight is associated with each of these based on complexity (simple, average, complex).
- The unadjusted function point (UFP) count is computed by multiplying each count by the weight and summing all values.
- UFC = $\sum$(number of elements of given type) X (weight)
- Function point count further modified by complexity of the project, 14 value adjustment factors (VAFs).

Function Point Analysis (FPA)

- Function points are a standard unit of measure that represent the functional size of a software application.
- FPA is a method to estimate costs based on counting the number of different data structures that are being used.
- In the FPA method it is assumed that the number of different data structures is a good size indicator.
- FPA is particularly suitable for projects aimed at realising business applications.

FPs are Useful:

- As requirements are the only thing needed for function points count, can be estimated early in the project.
- As they are independent of technology and programming languages.
- In estimating overall project costs, schedule and effort by creating ratios with other metrics such as hours, cost, headcount, duration and other application metrics.
- As they quantify and assign a value to the actual uses, interfaces and purposes of the functions in the software.
- Low cost, adding the process of counting FPs to a development process results in a cost increase of only 1%.
- Multiple function point counters can independently count the same application to within 10% accuracy of each other. Repeatability is important for comparisons.

FPs Drawbacks

- FPA is subjective
  - When calculating UFP count.
  - When assessing technology factors.
- There are problems with double counting. It is possible to account for internal complexity twice in weighting the inputs for the UFP and again in the VAFs.
- FPA applies well only to systems, which are dominated by input-output and filing functions.
- Because function points largely ignore internal processing complexity, care must be taken in applying FPA to complex applications, even in the business area.
- Different companies will calculate function points slightly different, making comparisons questionable.
- Hard to map function points to individual work packages for individual programmer productivity.

Uses

- The ability to accurately estimate COCOMO:
  - project cost
  - project duration
  - project staffing size
- An understanding of other important metrics, such as:
  - Project defect rate - errors per FP or defects per FP
  - Cost per FP - € per FP
  - FPs per hour / person month
  - Pages of documentation per FP
  - The productivity benefits of using new or different tools

Length Lines of code (LOC)

- Source lines of code (LOC) is a software metric used to measure the size of a software program by counting the number of lines in the text of the programs source code.
- LOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.

LOC are Useful:

- Since it is a physical entity; manually counting effort can be easily eliminated by automating the counting process.
- Serves as an intuitive metric for measuring the size of software because it can be seen and the effect of it can be visualized. Function points are said to be more of an objective metric which cannot be imagined as being a physical entity, it exists only in the logical space.
- In estimating overall project costs, schedule and effort by creating ratios with other metrics such as hours, cost, headcount, duration and other application metrics.
- Many existing software estimation models use LOC or KLOC as a key input.
- A large body of data and literature predicated on LOC already exists.

LOC Drawbacks

- Effort is highly correlated with LOC, functionality is less so. Skilled developers may be able to develop the same functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. Penalise shorter programs developed by more experienced developers
- LOC is a poor productivity measure of individuals, a developer who develops only a few lines may still be more productive than a developer creating more lines of code - even more: refactoring is likely to get rid of redundant code and keep it clean reducing the LOC.

LOC Drawbacks

- Language dependent
  - Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written in a language like COBOL. The number of FPs would be exactly the same. The LOC needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different.
- Lack of Counting Standards
  - There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. Though organisations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice especially in the face of newer and newer languages being introduced every year. Also different companies may count slightly different, making company comparisons unreliable.

Comparing Projects

- Project defect rate - errors per KLOC or defects per KLOC
- Cost per KLOC
- Pages of documentation per KLOC
- KLOC per person month
- The productivity benefits of using new or different tools

Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. The premise is that complexity is related to the control flow of a program.

The complexity of a program can affect the time it takes to code, test and fix it.

Many software developers define program complexity as the cyclomatic number proposed by McCabe.

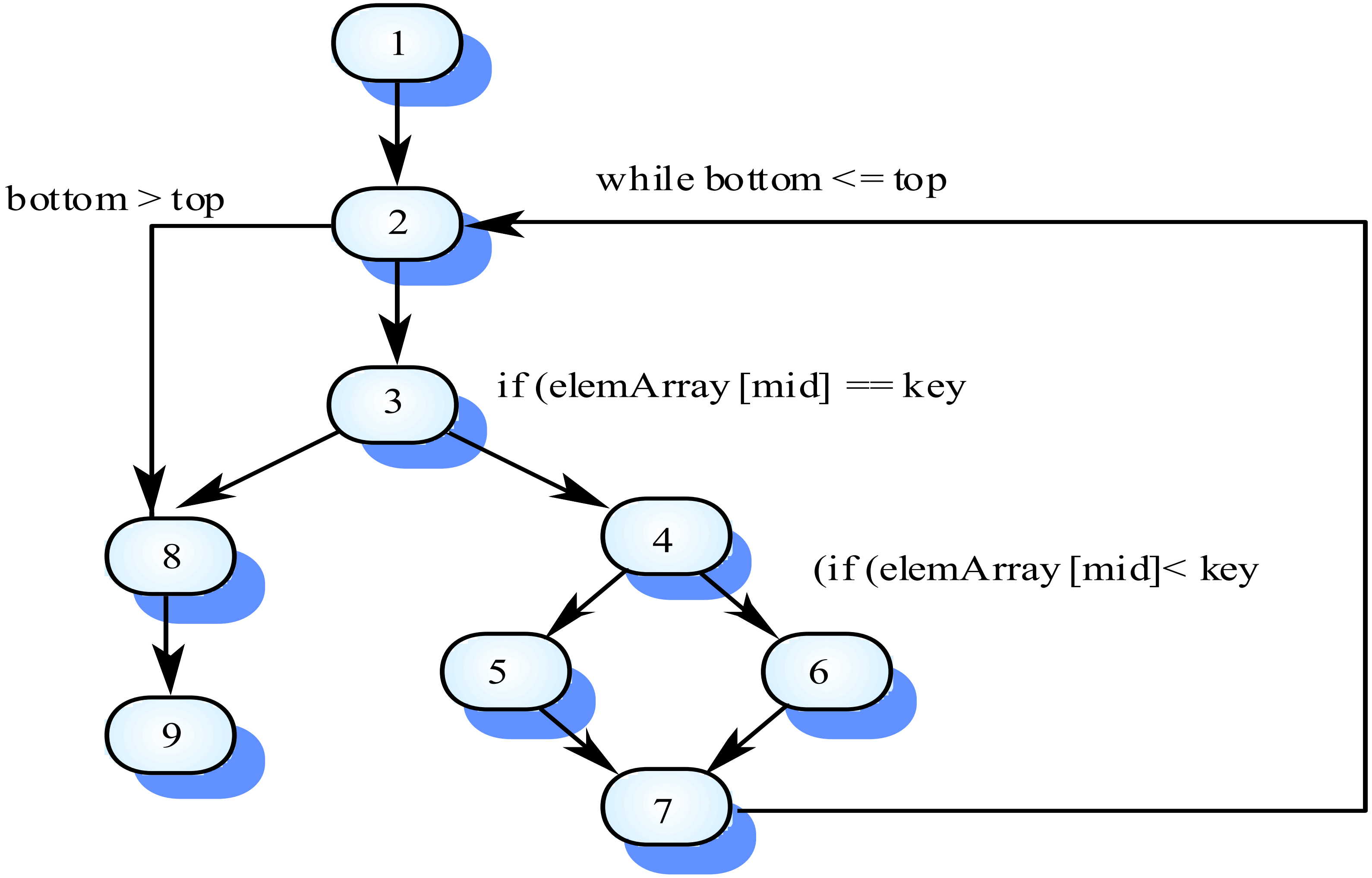
Cyclomatic Complexity

- It is a software metric that measures the logical complexity of the program code.
- It counts the number of decisions in the given program code.
- It measures the number of linearly independent paths through the program code.

Flow Graph

- A skeletal model of all paths through the program.
- Nodes represent decisions.
- Edges show the flow of control.
- Sequential statements are ignored (assignments, procedure calls and I/O statements). Each branch in a conditional statement is shown as a separate path and loops are indicated by an arrow looping back to the loop condition

Binary search flow graph

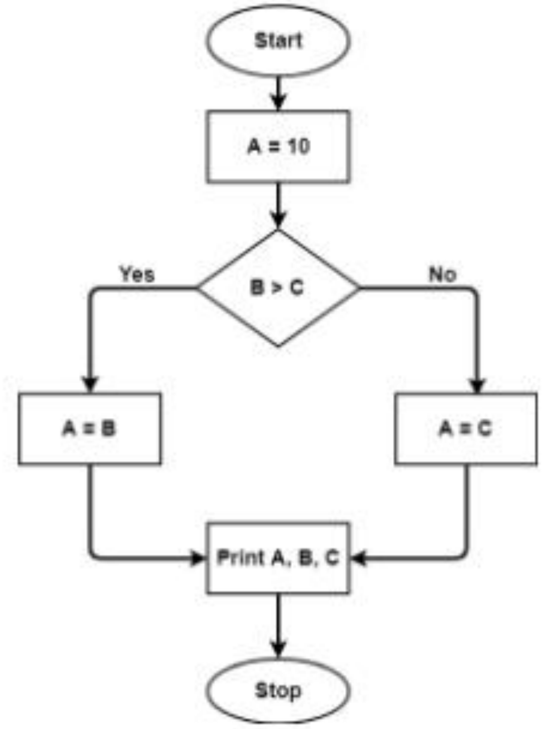


Cyclomatic complexity

- The cyclomatic complexity is computed as follows :
- CC= Number (edges) – Number (nodes) + 2
- CC = 11 – 9 + 2
- CC = 4

Exercise

```
A = 10
   IF B > C THEN
      A = B
   ELSE
      A = C
   ENDIF
Print A
Print B
Print C
```



CC= Number (edges) – Number (nodes) + 2

Cyclomatic Complexity

- Modules with high levels of cyclomatic complexity were most likely to be fault prone and unmaintainable.
- Mc Cabe proposed any module with a value greater than 10 should be redesigned.

Benefits of Reducing Cyclomatic Complexity

- **Reduces the coupling of code.** The higher the cyclomatic complexity number, the more coupled the code is. Highly coupled code cannot be modified easily and independently of other code.
- **Ease of understanding the code increases as the complexity decreases.** With a higher complexity number, the programmer has to deal with more control paths in the code which leads to more unexpected results and defects.
- **Ease of testing.** If a method has a cyclomatic complexity of 10, it means there are 10 independent paths through the method. This implies is that at least 10 test cases are needed to test all the different paths through the code. The lesser the number, the easier it is to test.

Cyclomatic Complexity Assessment

- Theoretically, it has been argued that the metric is too simplistic it fails to take account of data-flow complexity or the complexity of unstructured programs.
- A nested construct is more complex than a simple construct, but cyclomatic complexity calculates same complexity for both types of constructs.

Defects -based Quality Measures.

- Defect – A known error, fault or failure.
- A **failure** is something which is observed while the software is in operation.
- A **fault** is the part of a software document (code or other form of documentation) that is the encoding of human error.
- Failures are caused by triggering one or more faults with specific inputs. Faults will only cause failures if they are triggered with a particular combination of inputs.

Defects -based Quality Measures.

- A de facto standard measure of software quality is defect density.
  - **Known defects** that have been discovered through testing, inspection and other techniques;
  - Defect density = $\dfrac{\text{number of known defects}}{\text{product size}}$

Defect Density

- When using defect density for quality assurance purposes or to compare performance remember :
  - There is no consensus on what constitutes a defect. To make comparisons all parties must count the same way.
  - There is no consensus as to how to measure software size.
  - Defect density is derived from the process of finding defects. It may thus tell us more about this process than about the product quality itself.

Defect Density

- Even if the number of residual faults (those discovered after release) are known it can be hard to predict how the system will operate in practice. This is due to two key findings :
  - It is difficult to determine in advance the seriousness of a fault.
  - There is great variability in the way systems are used by different users, and users do not always use the system in the ways expected or intended. It is therefore difficult to predict which faults will lead to failures, or to predict which failures will occur often

Faults & Failures

- A small proportion of faults in a system can lead to most failures in a given time period; most faults in a system are benign and will not lead to failure in the same time period.
- It is possible to have a product with a large number of defects rarely failing.
- Finding and removing large numbers of faults may not improve reliability.

Reported Evidence

- Companies are reluctant to publish defect densities. Not withstanding the difficulty of determining the validity of figures there is some consensus.
- USA/Europe average is 5-10 per KLOC.
- Japan 4 per KLOC, may be misleading because only top companies reported.
- A defect density of below 2 per KLOC is considered good.