# The Importance of Life Cycle Modeling to Defect Detection and Prevention

**J.H. van Moll**
Philips Semiconductors
ReUse Technology Group
Prof. Holstlaan 4
5656 AA Eindhoven
The Netherlands
jan.van.moll@philips.com

**J. C. Jacobs**
Philips Semiconductors
ReUse Technology Group
Prof. Holstlaan 4
5656 AA Eindhoven
The Netherlands
jef.jacobs@philips.com

**B. Freimut**
Fraunhofer IESE
Sauerwiesen 6
67661 Kaiserslautern
Germany
freimut@iese.fhg.de

**J.J.M. Trienekens**
Frits Philips Institute
Eindhoven University of Technology
Den Dolech 2
5600 MB Eindhoven
The Netherlands
j.j.m.trienekens@tm.tue.nl

## Abstract

*In many low mature organizations dynamic testing is often the only defect detection method applied. Thus, defects are detected rather late in the development process. High rework and testing effort, typically under time pressure, lead to unpredictable delivery dates and uncertain product quality. This paper presents several methods for early defect detection and prevention that have been in existence for quite some time, although not all of them are common practice. However, to use these methods operationally and scale them to a particular project or environment, they have to be positioned appropriately in the life cycle, especially in complex projects.*

*Modeling the development life cycle, that is the construction of a project-specific life cycle, is an indispensable first step to recognize possible defect injection points throughout the development project and to optimize the application of the available methods for defect detection and prevention. This paper discusses the importance of Life Cycle Modeling for defect detection and prevention and presents a set of concrete, proven methods that can be used to optimize defect detection and prevention. In particular, software inspections, static code analysis, defect measurement and defect causal analysis are discussed. These methods allow early, low cost detection of defects, preventing them from propagating to later development stages and preventing the occurrence of similar defects in future projects.*

**Keywords:** Life Cycle, V-model, Defect Detection, Defect Prevention

## 1 Introduction

Due to the continuously growing possibilities provided by technology and their wider application, today's products are becoming more and more complex from a technical point of view. This is especially seen in (but is certainly not limited to) industrial products and consumer electronics. Products like these can typically be 'large' in terms of the number of sub-systems or components. A sub-system or component is best defined as a self-contained part serving a clearly defined purpose within the total product and having a clear interface to other components. In this paper, products comprised of multiple sub-systems and components are referred to as 'complex products'. Typical examples include medical imaging systems, electron microscopes, wafer steppers, broadcast and communication systems and satellite navigation systems. Development of such complex products is imposed with two constraints that make it particularly challenging. On one hand these products are increasingly confronted with higher quality demands while simultaneously being developed in the same or even shorter time spans. On the other hand, the development of complex technical products is itself a complicated undertaking resulting in equally complex organizational issues.

The increased quality demand forces development organizations to define, deploy and remember to employ adequate development and business processes to enable themselves to meet these demands. An adequate verification and validation process encompassing defect detection and prevention is an indispensable contribution to measuring and achieving an acceptable product quality level. The necessity of having effective and efficient defect detection and prevention is strengthened by the fact that the commercial lifetime of products, especially in consumer electronics, is decreasing. Each new product must provide more functionality and typically be developed faster than its predecessors where competitive behavior by producers of comparable products is making the time pressure even higher. This can only be achieved if the software is developed "right the first time", which calls for effective defect prevention or at least defect detection that removes defects as soon as they are introduced, so downstream rework cost and time are reduced.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

In order to tackle the complex organizational issues that arise from the development of complex technical products and deal with multi-component products in possibly multi-disciplined development environments, creation of these products is divided into 'manageable portions'. Typically, this takes the form of separate, and in most cases, concurrent development projects. These separate projects are each responsible for the implementation of a part of the product. Examples of concurrent sub-projects include hardware or software components and subsystems, applications, infrastructures or platforms. Decisions on how the exact divisions are to be made, may depend on various considerations [9]. Factors influencing the distribution include differences in local development costs, availability of expertise, competences or resources.

Applying a distributed development approach creates virtual, if not physical, distance between the parties involved [2]. One effect is that 'classic' project management, as used in single projects, is no longer adequate. There are far-reaching consequences for project planning and tracking in particular. The introduction of inter-project co-ordination or project alignment is caused by dependencies like the order of project deliverables, resource sharing or time constraints. Likewise, use of a simple development life cycle becomes questionable as well, as there is not one single project, but multiple concurrent ones. A life cycle defines a project's logical activity flow through identification, specification and structured composition of phases, milestones, deliverables and activities. The life cycle is necessary to implement the objectives of the project. Note that in this paper, the term 'life cycle' only addresses the creation phases of a product.

In this paper, we intend to bring the modeling of the life cycle and defect detection and prevention together. This is done by positioning a simple development life cycle as a point of departure in Section 2. It is shown how life cycles can be seen in case of complex development. Subsequently, Section 3 gives insight in how defect detection is different in complex environments compared to single project development. Problems actually encountered in real-life are used as examples in Section 4 to illustrate some typical effects of inadequately modeled life cycles. Section 5 then presents empirical findings that make the case for early defect detection and prevention. Next, Section 6 presents an approach for early defect detection by combining industrially proven techniques and discusses their tailoring using life cycle modeling. Section 7 concludes the paper.

## 2 Development Life Cycle

A basic form of development life cycle is the V life cycle illustrated in Figure 1. This is a simplified representation of the *Vorgehensmodell* (V-model) [10]. The V-model explicitly recognizes the testing activities throughout all phases of development.

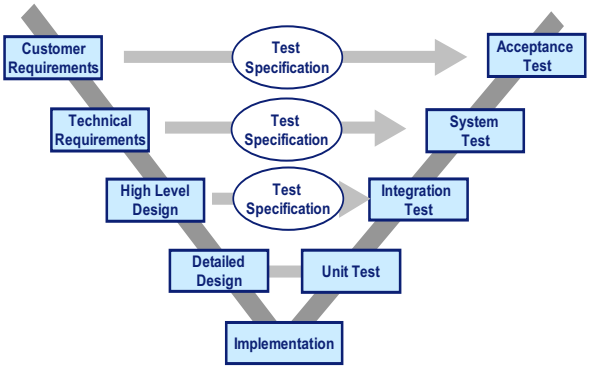


**Figure 1. The simplified basic V-model**

The V-model in the form above is often used for the creation of simple products developed in a single project. In practice, the phases will not be entirely sequential as shown. Some overlap of phases is possible, but is limited to a certain degree. However, in the development of the complex products discussed in Section 1, such simple life cycles are no longer appropriate. Therefore, we will now discuss more complex development life cycles.

A method commonly used to manage complex product development projects is layering of project management responsibilities. Typically there is one 'upper' layer of project management, often called 'system level' project management or 'top level' project management. Both names imply that this level is eventually responsible for delivering the final product. Responsibility for development of the product's components (i.e. sub-projects) is delegated to 'lower' levels of project management forming a hierarchy of related projects. This method tacitly assumes it is possible to decompose a product into clearly distinct components that can be developed in individual projects. In fact, a project hierarchy is more or less a translation of the product's main architecture as shown in Figure 2.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

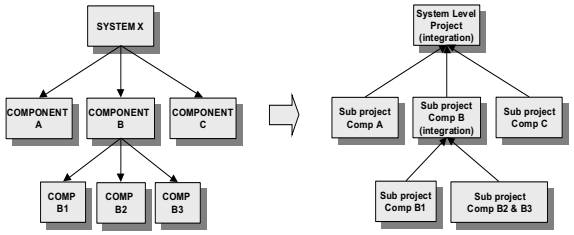


**Figure 2. Translation of system architecture into project hierarchy**

Before the architecture of a complex product is fully understood, preliminary architecture studies typically provide sufficient information for project management and project hierarchy design. When deciding which components should be developed in a separate project, care must be taken not to 'make the resolution of the decomposition too high'. In general, it makes no sense run a separate sub-project for a cluster of activities with a total effort of two man-weeks.

The simple life cycle in its default form presented in Figure 1 is inadequate for representing development of complex products. Figure 3 is a modified representation of Figure 1 showing how complex product development can be seen from a life cycle point of view. The single V-life cycle is horizontally 'cut in half' to distinguish a top-level system and a components layer.

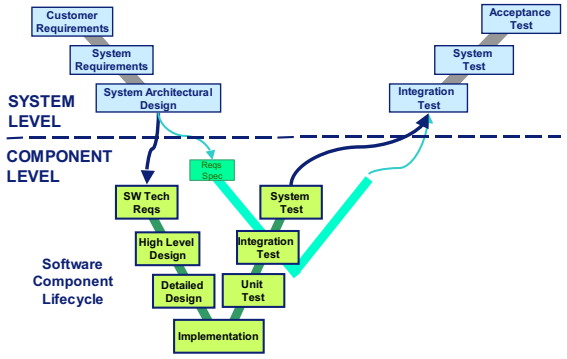


**Figure 3. Complex development life cycle**

Components identified in the system architectural design and can be of many types. Typical types include software sub-systems, hardware sub-systems, firmware or any combination of these. Once identified as sub-projects, development of the components takes place according to the components' individual life cycles. Each type of component requires its own type of life cycle. Clearly, hardware development requires a different type of development life cycle than software development.

In Figure 3, the life cycle used by the software component development project is again a single V-life cycle. It is however only a partial life cycle as it does not include an acceptance test against the customer requirements. It begins by defining software technical requirements relevant to development of the software component.

Other components all use some form of an incomplete life cycle. Eventually, the individual components are available and are integrated to create the complete product. This is where the deliverables of separate component life cycles come together in the Integration Test of the top-level system life cycle. Note that life cycle phases are similar on different levels. For instance, there is a system test in the top-level system life cycle but there is also one at the component level. It is therefore very important to be specific when talking about test levels.

Life Cycle Modeling, as defined in this paper, allows an early overview of the overall development life cycle structure as shown in Figure 3. Life Cycle Modeling is best described as the process of adapting a default or standard development life cycle to suit a project-specific context. The result is an instance of a development life cycle defining applicable milestones, phases and activities that take the actual project-specific circumstances into account. The modeled life cycle is used as the primary basis for project planning, tracking and control.

Life cycle modeling becomes a particularly critical issue in development of complex products with a multi-disciplinary and/or multi-site environment. In these environments, multiple projects are often executed concurrently requiring multiple engineering domains to confront each other with their different ways of working. Also, domain-specific development approaches, employed either by hardware, software, mechanics or other disciplines have mutual differences. Despite the existence of detailed life cycle descriptions from several sources (e.g. [6], [10]) and associated guides or standards on how to use them in actual practice (e.g. [5]), life cycle modeling as such is often not given the attention it deserves.

To keep product implementation manageable, it is of extreme importance to project management at the overall system level to plan and track meticulously. There are typically many dependencies between projects to be managed such as projects needing each other's deliverables.

Paying serious attention to adequate Life Cycle Modeling at the beginning of a new development project is indispensable:

- to get a clear view on the development life cycle of the overall system



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

- to become aware of the various dependencies between development life cycles used in related projects.
- to recognize the existence of these dependencies and to structure the project hierarchy accordingly (alignment of the life cycles)
- to come to an overall life cycle that is suitable as a basis for project planning, tracking and control

As a matter of fact, Figure 3 is highly simplified and would typically be much more complicated for any complex product in reality. Component life cycles can in turn instantiate other life cycles for sub-components, resulting in 'nested' life cycles. Incremental development would result in a chained sequence of partial life cycles. The complete life cycle for a complex product might even contain multiple levels of nesting and/or chaining.

When modeling a life cycle, it must be explicitly noted that whatever the project context may be, there is no such thing as *the* life cycle. This is completely analogous to the idea that for any system, there is no such thing as *the* architecture. Just as design decisions determine a system's architecture, modeling decisions determine a project's proposed life cycle. And expert judgment is needed to make the most effective decisions.

## 3 Relation between Life Cycle and Defect Detection

The development of a complex product can be considered as an ordered sequence of phases and activities, regardless of the specific development approach (e.g. incremental, waterfall, evolutionary). In theory, each phase and each activity is open to the injection of defects in the resulting information or work product by many causes (Figure 4). Defects injected add to those already injected in previous phases and might propagate to subsequent phases.

In order to develop the system "right the first time", the number of injected defects should be minimized, which is the task of defect prevention measures. In reality, defects will occur. Therefore, for any given phase, defect detection measures should be taken. The measures must be appropriate to the typical type of defects injected and the information or work product produced. The goal is to minimize the amount of defects that propagates to the subsequent phases. Ultimately, the number of residual defects in the end product should be as low as possible.

Defect detection activities in a certain phase can focus entirely on finding defects injected during that phase and in work products or information produced within the single project. In addition, they might also detect defects injected in previous phases.

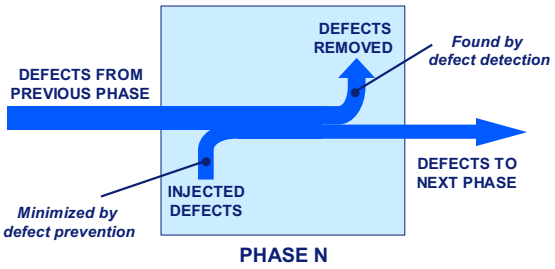


**Figure 4. Defect injection and detection in a phase**

In the development of complex products, multiple projects each have their individual life cycle, each with its own phases. This causes defect injection to become dispersed over multiple projects. However, to minimize the number of residual defects in the final product, defect detection should start as early as possible to prevent defects from propagating. But as a consequence of the phases being in different projects, defect detection is done by different parties. This introduces risks like the inadequate review of work products, occurrence of 'blind spots' with respect to test coverage or over-testing of components. Life cycle-wide co-ordination of defect detection is therefore needed to ensure effectiveness and efficiency of defect detection activities.

Guidance and co-ordination of defect detection should be provided by a defined test approach and accompanying test strategy. Test approach and test strategy go hand in hand. The test approach is to determine *what* is to be tested in terms of a product's functional characteristics and quality characteristics, *when* during the overall development life cycle and by *whom*. The test strategy is to determine *how* things will be tested including the test techniques used and desired test depth. When multiple levels of testing are done by parties external to the overall system development project, the test approach should be addressed in a master test plan [1].

In complex system development, there are typical areas where the likelihood of defect injection is significantly higher compared with the development of simple (i.e., single part) systems. These areas have in common that they are all situated at the 'transitions' between related life cycles (i.e. the dotted line in Figure 3). At these transitions, information or work products flow cross the boundary between one life cycle and another. For example, information generated in the system project is transferred to several separate component development projects. There, the information is transformed into other types of information needed to develop the individual components.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

Complex system development is very likely to contain many of these life cycle transitions. Significant parts of the total development take place in component development projects outside the direct scope of the system project. Life Cycle Modeling is an indispensable first step in localizing defect-sensitive areas in the overall life cycle. This is because the purpose of Life Cycle Modeling is to gaining a good view on the overall system development life cycle and its relation to and interaction with (sub-)projects. By thinking about how projects are to be aligned, project management is forced to think about the flow of information and work products within and between projects (e.g. requirements specifications, intermediate deliverables, test objects). This thinking exposes defect-sensitive areas and enables the appropriate development and test phases to be included into the life cycle. It allows for a timely application of appropriate defect detection measures at the right moment by the right parties during the entire development process.

## 4 Real-life Problems

An investigation of a number of past complex development projects, executed in the period between 1999 and 2001, was carried out by the authors to determine the most urgent problems actually experienced with respect to defect detection. The investigation included over twenty projects (with effort-sizes between 5 and 30 person-years) involving hardware, embedded software and firmware development in the area of industrial and consumer electronics by companies in the Netherlands and Belgium. The problems found involve a number of practices often applied in complex and software-intensive product development and illustrate a relationship with the life cycle used. A clear view of the life cycle beforehand would have been a significant contribution towards preventing the occurrence of the problems mentioned.

*Reuse of existing components*

Reuse is a way to reduce total development effort or to minimize product quality risks by using an already existing and tested component with known specifications and behavior. A mistake very often made during project planning was to not include any phases or activities to cover the effort needed for proper reuse of the component [3]. This means that a component was taken ‘as is’ and treated as if it were an off-the-shelf product. In project planning the reuse was seen only as the reused component being an input deliverable at a point in the overall life cycle.

As a consequence, a test phase for the component taking different test conditions into account was not scheduled. No effort at all was planned for determining whether the test conditions originally used during testing of the component were still valid in the target environment of the current application.

Problems with the reused component were detected only during integration. A separate test of the component still had to be carried out to isolate problem causes. An immediate result was that additional, unplanned effort had to be spent on these activities before integration could be completed successfully. The start of the subsequent system test was further delayed which jeopardized the project’s time schedule.

*Application of third-party components*

Third-party components are existing components supplied by organizations commercially exploiting them. Using third-party components is a form of reuse. One of the differences being that the provider of the component is located outside the own company. Third-party components were often considered to be usable "as is". However, components like these are typically generic and parameterized enabling the provider to sell the component to different customers. This implies that customization or tailoring of the component is needed to suit specific applications. The effort needed to do so was often not planned.

Information on functional and behavioral characteristics of a third-party component, like detailed test reports, can be very difficult or impossible to get from the provider. However, this kind of information can be crucial to other parts of the system developed in house. Unless it is absolutely clear how the component is to be used, additional phases to study or pre-test a third-party component should not be forgotten and should be included in the overall life cycle. However, the performance of a pre-test is rarely practiced. As a consequence, system projects were often confronted with problems while integrating the component.

*Development by external organizations*

Today it is common practice to leave the development of components to separate development teams within the organization or to organizations external to the company (subcontractors). Whatever their organizational relationships with the overall system project, these teams always need input from the system project. Likewise, there is always a moment in time where they deliver products back to the system project. Very often these moments in time appear in project planning data as the ‘boomerang’ construction.

This is called a ‘boomerang’ because the overall system project has typically planned a moment in time where it feeds another internal or external project with input such as requirements specifications (throwing the boomerang). Likewise, there is a moment where the



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP’02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

overall system project plans to accept the result of the other project's effort (catching the boomerang). The period between providing the input and receiving the output is generally only assigned a duration. With no intermediate checks included in the overall project planning, it is not clear what is happening with the boomerang in its flight. Any problems related to the component appear only at the time of, or just before, acceptance of the component by the system project. It is extremely difficult for the system project to anticipate on the quality of the component (to catch the boomerang, can I hold my position or should I take a few steps to the left or to the right?). Though boomerangs mainly appear in the case of development by external organizations, they were also seen in case of development done by teams within one company.

The main problem with boomerangs is that, in their purest form, they do not take the component's development life cycle into account. When simply handing over the system's requirements specifications it remains uncertain whether the requirements will be interpreted correctly by the component development team unless this is verified in some way. By explicitly considering and including a component's life cycle into the overall system life cycle, planned verification is made possible.

For example, it can be planned that system architects participate in the review of component requirements specifications created by the component development teams. Any defects in the requirements, and certainly those caused by misinterpretation of source requirements, can be removed immediately so they do not result in problems during testing of the component later [11], [12].

System project management is sometimes aware of the dangerous effects caused by reduced visibility of a component's quality. To overcome this, system project management required a test report to be included in the component's delivery. Requiring such a test report is in itself a good practice as the report can aid in determining the component's quality. In many cases there was an acceptance test of the component, but it makes no sense to learn what has been tested and how only when the component is delivered. It frequently turned out that the test strategy for the component did not match with the strategy defined for the overall system. Extensive testing had been done on the component, which perhaps would have been easier once integrated in the entire system. Or the component had been extensively tested, while at the same time test cases having the same test focus and test goals were being specified at the overall system level.

## 5 Early Defect Detection and Prevention

The investigation of the real-life problems learned that complex development projects applying the practices above often complained about 'problems with testing'. More specifically the projects were experiencing that:

- integrating the necessary components was highly problematic and resulted in an unacceptable amount of rework effort
- tests at the various levels in the projects' hierarchy were not aligned, causing 'blind spots' with respect to the test coverage of the final product's requirements
- defects were detected too late and should have been detected by other parties or at lower test levels

Many companies already struggle with their testing process even in single level development projects. These testing related problems are aggravated in complex product development. The life cycle can help in by recognizing and creating visibility into defect-sensitive areas so defect prevention actions can be initiated. A main objective of defect prevention measures is to prevent defects from being injected in the first place (Figure 4). Like defect detection, prevention should be coordinated in complex product development as multiple parties are involved in applying the measures. Defect prevention actions as well as actions for early detection of defects address especially the third problem mentioned above, the late detection of defects.

The importance of including defect detection and prevention activities into the development life cycle is illustrated by several empirical findings. Boehm and Basili [4] present a top-10 list of observations highlighting both the necessity of defect detection and prevention as well as techniques to help avoid defects in order to master software's complexity and accelerated schedules.

They observed that "finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase" and "current software projects spend about 40 to 50 percent of their effort on avoidable rework". These observations illustrate that current projects, simple projects but particularly complex projects, can benefit significantly by preventing defects or at least detecting them as early as possible. One solution to address these observations is to include appropriate defect detection activities into the life cycle in order to detect defects as early as possible. Typically, software inspections are employed for that purpose [7].

Another observation of Boehm and Basili is that "About 80 percent of avoidable rework comes from 20 percent of the defects" and "About 80 percent of the



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

defects come from 20 percent of the modules and about half the modules are defect-free". As a consequence of these skewed defect distributions it can be concluded that focussing the defect countermeasures should pay off, in addition to early defect detection.

In order to learn from these observations we present an approach that consists of several methods that should be taken into account when modelling the defect detection and prevention into the life cycle of simple and complex projects. This approach is based on proven techniques that support the quantitative management of the inspection and test process.

# 6 An Approach for Defect Detection and Prevention

First of all, defect detection and prevention call for constructive and analytical activities. Constructive activities prevent defects from being introduced by using appropriate processes, methods, guidelines, etc. Analytical activities prevent defects from becoming too expensive or risky and prevent them from re-occurring. In order to prevent defects from becoming too expensive and risky, verification and validation activities such as inspections or testing are used. In order to prevent defects re-occurring, the defect root causes are analyzed and removed.

Our quantitative approach for defect detection and prevention uses the following four steps:

1. *Perform inspections and testing throughout the life cycle.*
   As discussed in the previous sections, the challenge here is to place inspections at transition spots between multiple projects in order to exploit the beneficial effects. The challenge with testing is to find an appropriate coverage of the product from both a cost-oriented and a quality-oriented point of view.
2. *Collect measurement data as a regular part of the inspection and test process.*
   The collected data required for the proposed quantitative inspection and test approach encompasses defect data such as the number and "type" of defects as well as data regarding the code and design structure.
3. *Analyze the data.*
   Analyzing the data for the purpose of defect detection and prevention is guided by the questions discussed below.
4. *Feedback and interpret data as a regular part of the process.*

The core of this approach is the analysis. With the proposed method the following questions are addressed throughout the analysis:

- *What kind of defects do we introduce most?* This question tackles the observation mentioned above that there are skewed defect distributions. Thus, an answer to this question will help us to focus defect countermeasures.
- *Do we detect the right kind of defects?* An answer to this question will help to detect defects as early as possible. If defects of an unexpected kind are detected, this might indicate problems with the inspection and test approach that can get worse if not tackled by appropriate countermeasures.
- *Where are defects most likely located?* Similar to the first question, an answer to this question allows us to focus inspection and test activities on those components where they are most likely to pay off.
- *How can we prevent similar defects?* An answer to this question helps to learn from experience and prevent defects.

In order to analyze the data and answer these questions, we employ a set of practical, proven techniques, namely defect classification, defect causal analysis, and static code analysis. In the following subsections, each of these techniques is described in more detail.

## 6.1 Defect Classification

Defect classification is typically used to answer the question "*What kind of defects do we introduce most?*" Defect classification schemes are used by organizations of low and high maturity. For example, Paulk et al. [8] report that many high maturity organizations (CMM level 4 and 5) use defect classification (esp. orthogonal defect classification) for their quantitative management. But also when implementing measurement programs in companies with lower CMM-levels, defect classification is used very frequently (e.g. [13]).

Depending on the exact purpose of data analysis, various defect attributes can be captured in a defect classification scheme, such as location (i.e., where in the system the defect was detected), timing (i.e., when was the defect inserted), symptom (i.e., what was observed when the defect surfaced), and cause (i.e., what error led to the defect) [14], [20], [25].

Displaying such defect data in a simple bar chart, as shown in Figure 5, allows easy detection of defect clusters and focus of defect measures.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

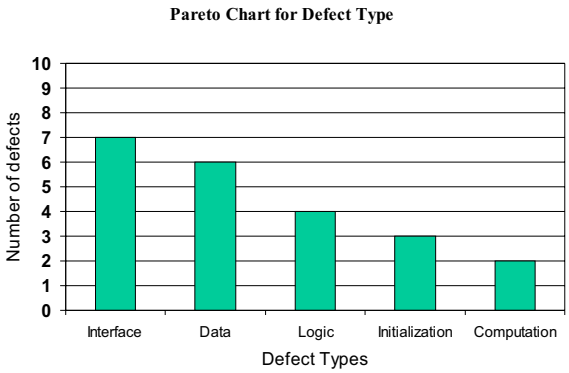


**Figure 5. Example of defect classification results**

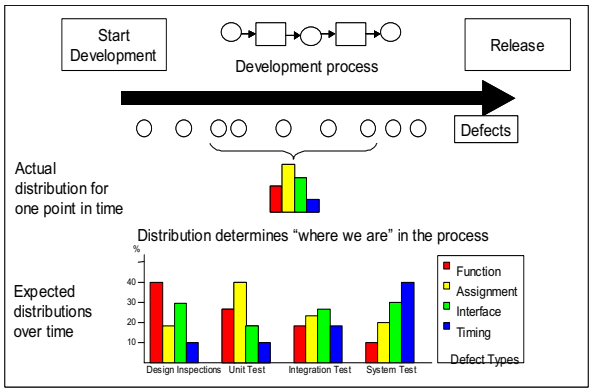


**Figure 6. Signatures of ODC Attribute Defect Type**

One specific defect classification scheme is the Orthogonal Defect Classification (ODC) Scheme [24]. One purpose of this scheme is to give project teams feedback on the progress of the current project, and thus enables us to answer the question "*Do we detect the right kind of defects?*" The ODC Scheme consists of a total of eight attributes. The attribute *Defect Type* plays a crucial role in the scheme.

*Defect Type* classifies the fix that was made to resolve the defect. An interesting aspect was taken into account when developing the set of attribute values: For each value of Defect Type an expectation exists as to which detection activity (e.g., unit test, function test, system test) defects of that attribute value should be detected. If an analysis of the defects reveals that activities are not finding the right types of defects, these processes obviously need to be improved [16].

For example, defects of type Function are those requiring a formal design change. It can be expected that the number of Function-defects decreases over time. This profile of an attribute value over time (i.e., different detection activities) is called a signature. Based on such a signature, deviations from the expected trend can be identified. For example, if the number of Function-defects is increasing with testing time, then a problem might exist that should be investigated.

Since each attribute value has a signature of its own, the detection activity has a specific expected distribution of attribute values. The progress of the project can be measured against these expected distributions, as shown in Figure 6.

## 6.2 Defect Causal Analysis

Defect Causal Analysis [23] or the Defect Prevention Process [19] are techniques typically employed to answer the question "*How can we prevent similar defects?*". The idea of Defect Causal Analysis (DCA) is to analyze defect data to find a systematic error. In this context, a systematic error is an error committed repeatedly and is therefore causing many defects and failures. Once the systematic error is identified, its underlying cause can be identified and appropriate process changes will prevent the error as well as the resulting defects and failures. Overall, this technique aims to prevent defects or at least enable an earlier detection.

One important part of this technique is the identification of the systematic error. For this purpose, the analysis of defect classification data is used. Based on a Pareto-Chart, the type(s) of defects occurring most often are identified. These types of defects typically hint to the systematic error.

Depending on the actual implementation of Defect Causal Analysis in an organization, either defect reports of a sample of defects are analyzed in a qualitative manner [23] to identify the systematic error or the identified types are used directly for reasoning about the systematic error [15].

An example from [15] is shown below as an illustration. The defect data were collected during maintenance in a Hewlett-Packard division developing systems software (using the language C, SA/SD design, informal and sporadic code inspections, branch coverage testing, regression testing). The defects were classified according to the HP defect classification scheme [15]. Figure 7 shows the defect classification data (also weighting defects according to their cost).



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

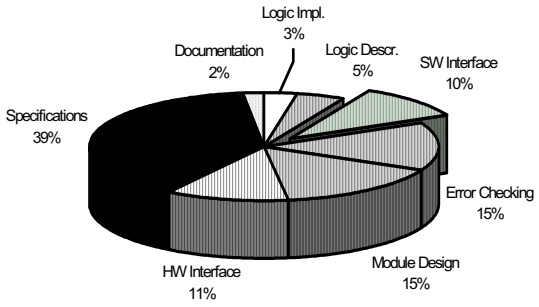


**Figure 7. Normalized defect data**

This figure shows that specification defects make a large proportion and therefore offer much improvement potential, especially for new products. Consequently, it was decided to prevent specification-defects in the future.

From the type of defects (i.e., specification) the systematic error was not obvious, so a brainstorming session was performed in the course of which a Fishbone-Diagram was developed. As root causes, the following were identified: unclear understanding of customer segments, lack of clearly assigned responsibilities, absence of document standards, and several changes in the hardware. To improve the process (and prevent future defects) it was decided to have the marketing department set up customer visits in order to learn about customer needs. Configuration management responsibility was assigned to one person who selected a tool for version control. And a task force was set up to develop a new specification documentation format. This example is narrated in a straightforward manner. In practice, however, several attributes are typically investigated in an explorative manner to investigate a meaningful subset of data.

Often, the crucial attribute(s) used in the course of a Defect Causal Analysis is the attribute or attributes capturing the cause of a defect. Such a classification (e.g. [17] or [19]) aims at more directly pinpointing to the root causes.

### 6.3 Static Code Analysis

In order to answer the question "*Where are defects most-likely located?*" an applicable technique is static code analysis. This technique allows up-front identification of components that are defect-prone or difficult to maintain. These are components that should be inspected or tested with high priority. Thus, inspection and test activities can be focused on those parts of the system that most likely need the attention.

The rationale behind Static Code Analysis (SCA) is that structural properties of a component such as coupling or cohesion (i.e., internal quality attributes) are related to external quality attributes such as reliability and maintainability (Figure 8). Once such a relationship has been empirically demonstrated it is possible to predict, based on structural properties of a component, its reliability (e.g., defect-proneness) or its maintainability (e.g., maintenance effort). Since structural properties can be measured as soon as a component is completed, a prediction regarding its quality can be made early on.

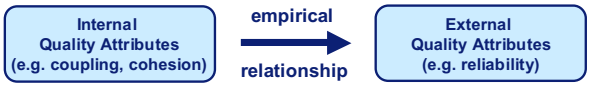


**Figure 8. Rationale for Static Code Analysis**

The general process of Static Code Analysis involves measuring the relevant structural properties from the components under study, performing a statistical analysis on the collected measurement data, and feeding the analysis results back to the development team. The statistical analysis can take two forms.

First, it is possible to use statistical methods such as regression analysis to predict, based on structural properties, the defect-proneness of a component or its likely maintenance effort [21], [22]. These prediction models can be used directly for decision-making.

Second, using a more pragmatic approach it is possible, once empiric relationships between structural properties and reliability and maintainability have been demonstrated, to measure the components under study and identify components that exhibit conspicuous measurement values for one or several structural properties [18]. These are components that should be subjected to further scrutiny. If no rationale for the normal measurement values can be found, countermeasures such a redesign, focused inspection or testing can be initiated.

An example for this approach is shown in Figures 9, 10 and 11. This example is taken from a Static Code Analysis performed in an industrial environment. In this context, control software for an industrial machine consisting of over 2000 classes was analyzed. For this purpose, the source code was analyzed with a tool that extracted selected structural properties (here: OO measures). Only those OO measures were considered for which an empirical relationship to reliability and maintainability had been demonstrated in previous studies [26],[27],[28]. Moreover, the measures were selected to be orthogonal to each other, (i.e., independent of each other). The measures used are shown in Figure 9.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

**Size Measures**
Effective Lines of Code
New Methods Implemented [26]

**Complexity Measures**
Inter-Class Method Invocation [26]

**Coupling Measures**
Import Coupling by aggregation [28]
Import Coupling by calls to ancestor classes [28]
Import Coupling by calls to friend classes [28]
OCAIC to a library class [28]
Import coupling by calls to library classes [28]
Export coupling by calls from other classes [28]
Export coupling by calls from friend classes [28]

**Inheritance Measures**
Number of Children [26],[27]
Depth of inheritance tree [26],[27]

**Figure 9. Static Code Analysis example: Measures**

The first step in analyzing the extracted data was to develop a distribution for each measure as shown in Figure 10. Such diagrams allow identification of classes exhibiting conspicuous measurement values for a given measure.

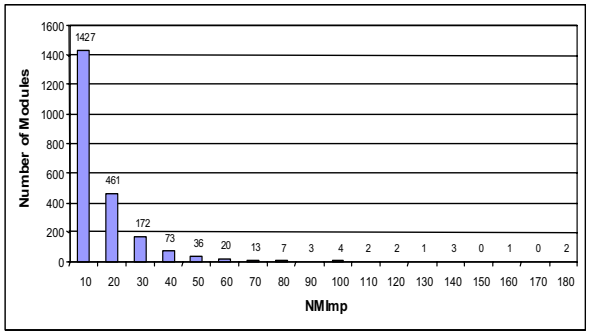


**Figure 10: Static Code Analysis example: Investigating one measure**

Next it was investigated which classes exhibited conspicuous measurement values for at least one measure. This was done using a table like shown in Figure 11. In the example, such analysis revealed 15 classes with conspicuous structural properties. Thus, it is possible to re-structure these classes in the future and to apply specific inspection and test activities to them.

| | Measures | | | | | | |
|---|---|---|---|---|---|---|---|
| Classes | OCAIC | AMMIC | IFMMIC | OCAIC-L | OMMI-L | OMMEC | FMMEC |
| C-1 | **33** | **75** | 0 | **74** | 20 | 0 | 0 |
| C-2 | **64** | 37 | 0 | **40** | 30 | 2 | 0 |
| C-3 | **37** | 1 | 0 | **49** | 6 | 0 | 0 |
| C-4 | **75** | 29 | 0 | **50** | 30 | 1 | 0 |
| C-5 | **45** | 1 | 0 | **55** | 9 | 0 | 0 |
| C-6 | 6 | **55** | 0 | 12 | 16 | **202** | 0 |
| C-7 | 2 | 29 | **12** | 0 | **50** | 0 | 0 |
| C-8 | **28** | 34 | 0 | **44** | 10 | 0 | 0 |
| C-9 | 0 | 25 | **14** | 0 | **52** | 0 | 0 |
| C-10 | 1 | **86** | 0 | 6 | 0 | 38 | 0 |
| C-11 | 5 | 22 | **93** | 11 | 7 | 6 | 0 |
| C-12 | 1 | 0 | 0 | **76** | 2 | 3 | 0 |
| C-13 | 6 | 2 | 0 | 19 | **122** | 5 | 0 |
| C-14 | 2 | 8 | 0 | 26 | 6 | **1523** | 0 |
| C-15 | 0 | 6 | 1 | 18 | 3 | 0 | **103** |

**Figure 11. Static Code Analysis example: Suspect classes**

## 6.4 Tailoring the Approach

The approach presented in this section combines a set of practical, well proven techniques for early defect detection and prevention. The combination of these techniques provides a comprehensive analysis framework for addressing the key questions of quantitative inspection and test management as posed in Section 6. However, in concrete environments a number of decisions have to be made in order to instantiate an approach that is most appropriate for the intended environment.

A good insight beforehand into the development life cycle is essential in making these decisions, especially for complex development projects. For example, these decisions concern which of the techniques presented in Section 6.1 to 6.3 should be implemented, when to perform the analysis, by whom and to whom the feedback is to be given.

Other decisions concern when to perform inspections, on which work products and by whom. A project-specific life cycle helps in identifying the mutual relationships between and the traceability of various levels of work documents that may be distributed over and created by multiple parties. As activities (like Defect Causal Analysis) contained in the analysis framework will cross the boundaries between development projects, responsibilities should be defined and coordination is needed. The adequacy of the activities and the extent to which coordination can be achieved highly depend on the degree of insight of the development life cycle at the time of instantiating the analysis framework for a given project.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

## 7 Conclusion

The development of complex products is a challenge in today's (software) industry. Complex products on one hand result in an equally complex organization of their development. On the other hand, such products have to meet increasing quality requirements in the same or even shorter development time spans. Thus, developing products "right the first time" is a key to success.

This paper addressed and combined two fields that can contribute in this respect, namely Life Cycle Modeling and Defect Detection and Prevention. While Life Cycle Modeling aims at mastering the complexity of today's (software) projects, Defect Detection and Prevention aims at developing the software right the first time. While each of these areas is beneficial to complex projects, it is their combination that makes a fast impact. This is due to the fact that the inspection and test process significantly benefits from a well-defined development life cycle as the primary precondition for application of the available detection and prevention measures. In practice, the inspection and test process is one of the first to experience project problems caused by a weak life cycle definition.

Development organizations typically spend a lot of effort conditioning and shaping processes used for requirements engineering, configuration management and project management. Though this focused effort may be justified and correct in making their way to higher levels of maturity, practice shows that life cycle modeling is still getting surprisingly little attention.

## References


[1] M. Pol, R. Teunissen and E. van Veenendaal. *Software Testing; A Guide to the TMap approach,* Addison-Wesley Pub Co., 1999

[2] J.D. Herbsleb and D. Moitra: "Global Software Development". *IEEE Software*, March/April 2001.

[3] D. Ribot, B. Bongard and C. Villermain: "Development Life Cycle WITH Reuse". *Proceedings of the 1994 ACM symposium on Applied computing*, Phoenix, USA, 1994

[4] B. Boehm and V. Basili, Software Defect Reduction Top 10 List, *IEEE Computer*, Vol.. 34, No. 1, pp 2-6, January 2001.

[5] *IEEE Std 1074-1997 Standard for Developing Software Life Cycle Processes.* IEEE, 1997

[6] *ISO/IEC 12207:1995 Software Life Cycle Processes,* International Organization for Standardization

[7] O. Laitenberger and J.M. DeBaud, "An Encompassing Life-Cycle Centric Survey of Software Inspection", *The Journal of Systems and Software*, vol. 50, no. 1, pp. 5-31, 2000.

[8] M. C. Paulk, D. Goldenson, and D. M. White, The 1999 Survey of High Maturity Organisations, Tech. Rep. CMU/SEI-2000-SR-002, Software Engineering Institute, Feb. 2000.

[9] C. Ebert and P. De Neve: "Surviving Global Software Development". *IEEE Software,* March/April 2001

[10] A. Bröhl and W. Dröschl. *Das V-Modell.* Oldenbourg Verlag, 1995

[11] D. Freedman and G. Weinberg: *Handbook of Walkthroughs, Inspections, and Technical Reviews.* Dorset House, 1990

[12] T. Gilb and D. Graham. *Software Inspection.* Addison-Wesley Publishing Company, Workingham, England, 1993

[13] B. Freimut, B. Klein, O. Laitenberger, G. Ruhe, "Measurable Software Quality Improvement through Innovative Software Inspection", *Proceedings of the ESCOM - SCOPE 2000*, pp. 345-353, 2000.

[14] B. Freimut, *Developing and Using Defect Classification Schemes*, Fraunhofer Institute for Experimental Software Engineering, Technical Report IESE 072.01/E, Kaiserslautern, 2001.

[15] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

[16] R. B. Kelsey, "Integrating a Defect Typology with Containment Metrics", *ACM Software Engineering Notes*, vol. 22, pp. 64--67, Mar. 1997.

[17] M. Leszak, D. E. Perry, and D. Stoll, "A case study in root cause defect analysis", in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 428-437, 2000.

[18] C. Lewerentz, F. Simon, F. Steinbrückner, H. Breitling, C. Lilienthal, M. Lippert: "External validation of a metric-based quality assessment of the JWAM Framework", *Dumke/Rombach: Software-Messung und Bewertung, Deutscher Universitätsverlag*, Wiesbaden, Germany, pp. 32 – 49. 2002.

[19] R.G. Mays, C.L. Jones, G.J. Holloway, and D.P. Studinski, "Experiences with defect prevention", *IBM*




Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE

*Systems Journal*, vol. 29, no. 1, pp. 4--32, 1990.

[20] P. Mellor, Failures, "Faults and changes in dependability measurement", *Information and Software Technology*, vol. 34, pp. 640--654, Oct. 1992.

[21] L. C. Briand, W. Melo, J. Wüst ; "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects", *IEEE Transactions on Software Engineering*, vol 28, no. 7, pp 706-720, 2002.

[22] L. C. Briand, J. Wüst, "Modeling Development Effort in OO Systems Using Design Properties", *IEEE Transactions on Software Engineering*, vol 27, no 11, pp. 963-986, 2001.

[23] David N. Card, "Learning from our Mistakes with Defect Causal Analysis", *IEEE Software*, pp. 56--63, Jan. 1998.

[24] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification -- A concept for in-process measurements", *IEEE Transactions on Software Engineering*, vol. 18, pp. 943-956, Nov. 1992.

[25] N. E. Fenton and S. L. Pfleeger, *Software Metrics - A Practical and Rigorous Approach*. International Thomson Computer Press, 2nd edition ed., 1996.

[26] L. Briand, J. Wuest, "Integrating Scenario-based and Measurement-based Software Product Assessment", *Journal of Systems and Software*, 59 (2001) p 3-22

[27] S.R. Chidamber, C.F. Kemerer, "Towards a Metrics Suite for Object Oriented design", in A. Paepcke, (ed.) *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, October 1991. Published in SIGPLAN Notices, 26 (11), 197-211, 1991.

[28] L. Briand, P. Devanbu, W. Melo, "An Investigation into Coupling Measures for C++", *Proceedings of ICSE '97*, Boston, USA, 1997.



Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP'02)
0-7695-1878-8/02 $17.00 © 2002 IEEE