THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CHALMERS |  UNIVERSITY OF GOTHENBURG

IMPROVING DEFECT MANAGEMENT IN AUTOMOTIVE SOFTWARE DEVELOPMENT

LIDEC—A LIGHT-WEIGHT DEFECT CLASSIFICATION SCHEME

Niklas Mellegård

Department of Computer Science and Engineering
Division of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY |
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2013

Improving Defect Management in Automotive Software Development
LiDeC—A Light-weight Defect Classification Scheme
NIKLAS MELLEGÅRD
ISBN 978-91-7385-906-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 3587
ISSN 0346-718X

Technical Report 97D
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology | University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers Reproservice
Göteborg, Sweden 2013

To my family:
Vivian,
Lo, Minoo,
Filip I,

And of course
Mom(s)
Dad(s)
Brothers

TABLE OF CONTENTS

PART VI—IN CONCLUSION

APPENDICES

ABSTRACT

Classification of software defects is a means to provide defect reports with a shared and well-defined structure. Quantitative analyses of the classification data facilitated by the shared structure are useful to industry practitioners and academic researchers. For practitioners, especially in large, complex or dynamic organizations, analyses can provide valuable information that characterize the development process, assist in identifying improvement opportunities, and provide one basis for predictions (e.g. product quality and resource needs). For researchers, classification data can facilitate evaluating effects of improved practices (e.g. new methods and tools) by analysing classified defect data before and after applying the hypothesized improved practice.

Although recognized as a promising approach, there has to date been limited research reported on defect classification schemes—specifically on the efficiency of their application in industry, and on the reliability of the classification data. Efficient classification is desirable as it minimizes the time required to classify defects. Reliability of the classification data is important as it directly affects the reliability of conclusions drawn from analyses of the data.

In this thesis, a defect classification scheme based on and compliant to the standard classification for software anomalies (IEEE Std. 1044) is described and evaluated. The classification scheme, LiDeC (Light-weight Defect Classification Scheme), was adapted to and applied in the development of automotive safety software.

Through case studies and an experiment, LiDeC was evaluated with respect to its industrial applicability, efficiency and reliability. The results show that analyses of classification data can provide new and useful information about the effectiveness of current development practices. Applying a classification scheme adapted to the target organization results in analyses that are more directly relevant to that organization. Academic experimentation showed that classification schemes are easy to learn and to apply—the experiment subjects were able to arrive at rational classifications, even though lacking domain specific knowledge.

The main contributions of the thesis include: the description of an adaptation of the standard classification scheme to a specific organization while maintaining standard compliance; an initial industrial evaluation of the applicability of the adapted classification scheme; a description of a methodology for comprehensively evaluating defect classification schemes; and finally, an investigation of current state-of-the-art with respect to defect classification, and a proposed roadmap for future research.

ACKNOWLEDGEMENTS

As there are far too many people who have contributed to the work behind this thesis—whether it be concrete results, inspiration, comfort, or support—to be enumerated, I should begin by apologizing to all those who are not mentioned in these acknowledgements.

Firstly, I would like to express my deepest appreciation for my supervisor Miroslaw Staron, and co-supervisor Fredrik Törner. As expected, there were many ups and downs, but thanks to your dedication and relentless efforts I have now finally reached the goal. I am also grateful to Thomas Arts, who recommended me for this position and was a great support before leaving the university to pursue other challenges. I would also like to express my gratitude to the senior colleagues at the department, especially Lars Pareto, Rogardt Heldal, Robert Feldt and Rikard Torkar.

During my PhD project, I have had the good fortune to have a close collaboration with Volvo Cars (VCC). Besides being an exciting environment to get insight into, I've met lots of brilliant people; many of whom have kindly taken time from their daily duties to support my inquiries. I would especially like to thank Hans Alminger and Jonas Ekmark. Hans was my first guide to the VCC maze, and Jonas helped facilitate many of my studies in the last half of the project. Both Hans and Jonas have spent many hours reviewing, and thereby certainly improving, my manuscripts before submission. Many thanks also to Jovan Gojkovic, Rune Solem and Daniel Söderström, who made some of the studies possible. Finally, to Peter Gergely who always seemed to have 5 minutes (usually turning into at least 30) for many interesting and inspiring discussions.

Thanks also to the additional PhD students in the ASIS project, Mohsen Nosratinia, Stina Carlsson and Henrik Lind, and to our two project managers Jacob Hägglund and Linda Wahlström. We had lots of ideas and lots of ambitions, some of which we were actually able to fulfil. As we had very different academic backgrounds and research interest, it cannot have been the easiest project to manage—"*like herding cats*" springs to mind (thanks to Ulrik Eklund).

To my fellow PhD students at the division, especially Håkan Burden, Ulrik Eklund, Kenneth Lind, Abdullah Al-Mamun (for introducing me to the Bangladeshi cuisine) and Ali Shahrokni; you've made life as a PhD student truly enjoyable. And, Håkan, sharing office with you has perhaps been the best part of my PhD studies, especially after you started leaving your gym shoes in the locker downstairs.

Most dearly, to my family which has during the course of these PhD studies grown from being just Vivian, Filip (the cat) and myself to also include Lo and Minoo—in addition to being the lights of my life and the reason I long back whenever I leave home, you have certainly forced me to use my working hours more efficiently.

Finally and most importantly, to Vivian: thanks for putting up with me during this very, very long journey. Hopefully I'll eventually find a way to make up for it.

The research leading to this thesis was conducted as part of the ASIS project (Algorithms and Software for Improved Safety), sponsored by The Swedish Governmental Agency for Innovative Systems (VINNOVA) under the intelligent Vehicle Systems (IVSS) programme, in collaboration with and executed at Volvo Car Corporation (VCC).

INCLUDED PUBLICATIONS

The main contribution of this thesis is based on five publications, included as chapters 5 to 9. The chapters have been kept as close as possible to the original publications (listed below). Minor modifications regarding language and layout have been made.

Chapters 1 to 4 and 10 have been added to provide the thesis with introduction, context, and a summary of results.

Chapter 5	Mellegård, N., Staron, M. *Characterizing Model Usage in Embedded Software Engineering: A Case Study* Published at 8th Nordic Workshop on Model Driven Software Engineering (NW-MoDE), Copenhagen, Denmark 2010
Chapter 6	Mellegård, N., Staron, M., Törner, F. *A Light-Weight Defect Classification Scheme for Embedded Automotive Software* Technical Report 2012:04, ISSN: 1654-4870, Chalmers University of Technology, Göteborg 2012
Chapter 7	Mellegård, N., Staron, M., Törner, F. *A Light-weight Defect Classification Scheme for Embedded Automotive Software and its Initial Evaluation* Published at 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE) Dallas, USA 2012
Chapter 8	Mellegård, N., Staron, M., Törner, F. *A Comprehensive Evaluation of Defect Classification Schemes* Submitted to Software Quality Journal 2013 (Ref. no: SQJO829)
Chapter 9	Mellegård, N., Staron, M., Törner, F. *Why Do We not Learn from Defects? Towards Defect-Driven Software Process Improvement* Published at International Conference on Model-Driven Engineering and Software Development (ModelsWard), Barcelona Spain 2013

ADDITIONAL PUBLICATIONS

The following additional publications have been written as part of the research for this thesis.

Mellegård, N., Staron, M. *Methodology for Requirements Engineering in Model-Based Projects for Reactive Automotive Software* Published at the Doctoral Symposium at the European Conference on Object-oriented Programming (ECOOP) Paphos, Cyprus 2008
Mellegård, N., Staron, M. *A Domain Specific Modelling Language for Specifying and Visualizing Requirements* Published at the First International Workshop on Domain Engineering, DE@CAiSE Amsterdam, The Netherlands2009
Mellegård, N., Staron, M *Use of Models in Automotive Software Development: A Case Study* Published at the First Workshop on Model Based Engineering for Embedded Systems Design, Dresden, Germany 2010
Mellegård, N., Staron, M. *Distribution of Effort Among Software Development Artefacts: An Initial Case Study* Published at Exploring Modelling Methods for Systems Analysis and Design, Springer Berlin / Heidelberg, Hammamet, Tunisia 2010
Mellegård, N., Staron, M. *Improving Efficiency of Change Impact Assessment Using Graphical Requirement Specifications: An Experiment* Published at the 11th International Conference on Product-Focused Software Process Improvement (PROFES), Limerick, Ireland 2010
Nosratinia, M., Lind, H., Carlsson, S. and Mellegård, N. *A holistic decision-making framework for integrated safety* Published at IEEE Intelligent Vehicles Symposium (IV), San Diego, USA 2010
Mellegård, N., Staron, M., Törner, F. *Identifying Improvement Issues and Best Practices in Introducing MDE: An Industrial Case Study* The paper, an extension of chapter 5, is in preparation for a journal submission.
Rana, R., Staron, M., Mellegård, N., Berger, C., Hansson, J., Nilsson, M., Törner, F. *Evaluation of standard reliability growth models in the context of automotive software systems* Published at the 14th International Conference on Product-Focused Software Process Improvement (PROFES) Paphos, Cyprus 2013
Ferwerda, A., Lind, K., Heldal, R., Mellegard, N., Chaudron, M. *Effects of Introducing MD*/DSL in Software Maintenance - A Longitudinal Industrial Case Study* Submitted to the 36th International Conference on Software Engineering (ICSE) 2014

PERSONAL CONTRIBUTION

For all publications above, the first author is the main contributor. In all publications appended in this thesis, I was the main contributor with regard to inception, planning and execution of the research, and writing. The same applies for the additional publications in which I am listed as first author.
For the three publications in which I am listed as co-author, the following contributions were made by me:

- *Nosratinia et al.*
  The publication summarizes the research goals of the joint research project ASIS (Algorithms and Software for Improved Safety) in which the present thesis was one of four parts. In the publication, I contributed to the inception of the publication, and was the main author of the introduction, and the sole author of section 2.
- *Rana et al.*
  The publication evaluates a number of reliability growth models by applying them to defect data from a real software development project. My contribution was to assist in the inception of the study, to provide the data, and to review an initial draft of the publication.
- *Ferwerda et al.*
  The publication is based on the thesis for the degree of Master of Science by Adry Ferwerda. Although based on the thesis, the majority of the publication was rewritten; mainly to shorten and make it clearer. I contributed to the inception, planning and writing of the publication.

PREFACE

On the final day of the 23rd IEEE Symposium on Software Reliability Engineering in November 2012, there was a panel debate titled "*What you Always Wanted to Know about Software Reliability but Were Afraid to Ask*". Instead of focusing on a particular topic, the panel encouraged an open discussion on all things related to software reliability. At one point, the panel came to discuss that one current and general problem in the software development industry is a lack of overview; for instance, whether there are issues common among organizations with respect to the reliability of their software. While there may be many opinions on such matters, there is a distinct lack of more objective evidence. The panel stated that, in part, the reason is organizations' reluctance to sharing defect data. Without access to such data from a large selection of companies and products, it is difficult to identify patters that might indicate common problems.

During the discussion there was an interesting—and, I think, highly relevant—anecdote told by a professor emeritus from the Polytechnic Institute of New York. I would like in this preface to relay my recollection of that anecdote.

According to the professor, there had been a tremendous increase in mass production in the US during and after World War I—from consumer products to infrastructure projects, such as railroad and bridges. In the following decades, there were a number of rare but catastrophic failures; e.g. collapsing bridges and buildings. A commonality among these cases was that there was not a single point of failure, but rather a series of faults that together caused the failure. Each incident appeared unique but overall it seemed systematic—rarely, but regularly, large structures collapsed with catastrophic consequences. Furthermore, in these construction projects there were many different companies collaborating, but the causes of the failures were too complex for one company to analyse single-handedly. Still, the question how to prevent such issues was high on the agenda.

In an attempt to address the problem, several of the concerned companies formed a consortium with the aim to collect and share defect data in what—according to the anecdote—came to be known as the "black book". The black book contained a variety of aspects of defects and their causes, from material fatigue to construction and manufacturing issues. As the black book contained a wealth of data from a large number of different products and organizations, it provided researchers with a much wider perspective than what would have been possible if a single organization had been studied. The data contributed over several years to detect complex and systematic issues,

and thereby improved both the quality of products and the efficiency of development.

In a sense, the software development industry is currently in a similar position as the construction and manufacturing industries were before the black book; there has been a recent and substantial increase in use of software, yet there does not seem to be a unifying ground to compare issues between organizations or even between departments within an organization. An initiative similar to the black book might be an approach to establish such a common ground; for instance to evaluate the impact of a particular type of process model or test method. In order to do so, however, there is a need to investigate what type of data that should be collected to be useful, while also satisfying individual organizations' need for confidentiality. The question is: (how) can that be done?

While it is not the intention in this thesis to provide a comprehensive equivalence of the black book, it does propose and rigorously evaluate an approach that can provide initial stepping-stones towards such equivalence. This is elaborate on in Part V of this thesis.

I

PART I—INTRODUCTION

Good luck is science not yet classified;
just as the supernatural is the natural not yet understood

— Elbert Green Hubbard

1 INTRODUCTION

In Hubbard's quote above, the concept of *good luck* is assigned factors that contribute to a successful outcome, and that are not known or fully understood. While good luck may certainly be a welcome addition to any engineer's work, it is inherently unreliable. Therefore, an engineer would prefer not to rely on good luck, but to understand and be able to control the factors influencing the outcome of a project. While it is the work of an engineer to apply knowledge about these factors in order to successfully enact a process, it is the work of a scientist to create knowledge by uncovering and explaining such factors.

The act of measuring aspects of a software development process is one way to facilitate the creation of such knowledge. It is through measurements that, for instance, the quality of products or the efficiency of development processes can be characterized, evaluated, predicted and improved (SEI, 2013). By characterizing, a deeper understanding—e.g. regarding processes, products and resources—can be gained and communicated. Evaluation enables, for instance, determining current project status with respect to plans, and assessing effects of changes to the process, methods or tools used. Predictions are important, for example to foresee future project needs, and thereby facilitating more precise resource planning, but can also allow early product quality assessment. Finally, characterizing, evaluating and predicting contribute to the identification of development bottlenecks and roadblocks, and thereby the identification of process improvement opportunities.

Much software engineering research with the goal to improve development efficiency and product quality consists of proposing new or modified practices (e.g. process models, methods and tools)—prominent examples include model-driven development (MDD) (Mellor et al., 2003), and agile process models (Korhonen, 2013). While such research is, no doubt, valid and valuable, it may often be difficult for practitioners to assess whether the benefits provided by the improved practices address the actual challenges faced by their specific organization. By establishing a system of metrics that can be used by researchers to evaluate and characterize the typical impact of new and improved practices, and that can be used by practitioners to characterize the current development organization, it would provide the practitioners with means to perform such assessments.

In practice, furthermore, the selection of improvement initiatives may often be guided mainly by expert opinion. While important, such opinion is naturally biased by the knowledge and prior experience of the experts, thus calling as a complement for evidence that is more objective. In the research project reported in this thesis, we initially (as reported in chapter 5) found ourselves in a similarly biased position, in that we set out to investigate how MDD could be used to improve the development of automotive safety systems—essentially "*equipped with a solution looking for its problem*". During that investigation, we encountered opinions from practitioners regarding which practices needed improvement, but also that they often found it difficult to get their voices heard within the organization (see chapter 9.2). The perceived reason was precisely a lack of hard data to back those opinions up. Consequently, the second half of the research project, and the main contribution of this thesis, was dedicated to examining how to obtain such data—thus effectively changing the research focus from "*how can MDD improve the development of automotive software?*" to "*how to identify improvement opportunities in automotive software development?*".

As with other engineering disciplines, there are in software engineering numerous sources of measurements. One such source is defects reported during development and after deployment. Although expected during development, defects can be regarded as a symptom of deficiencies in the development practices—patterns may indicate systemic process problems. For this reason, defect reports can contain interesting data; such as, how the defects were detected, what type of defects they were and what action was required to resolve it. When analysed quantitatively, patterns may be discovered that can be used to characterize, evaluate and predict various process activities; for example, information about the effectiveness of specific test activities or facilitate prediction of product quality. Defect

reports, however, are often written in free text, making quantitative analyses difficult. Defect classification schemes (DCSs) address this problem, by providing the defect reports with a shared structure, and thus enabling more efficient data collection.

Existing DCSs—such as ODC (Orthogonal Defect Classification) and IEEE Std. 1044—are generic and typically geared towards defects with an ultimate source-code manifestation. In addition, these DCSs typically target organizations that are in control of the entire development chain, especially the implementation phase. In the automotive software development domain, however, source-code is often developed by, and proprietary to, suppliers based on requirements and design specifications provided by the vehicle manufacturer (OEM[1]). The OEM, furthermore, is responsible for integrating the parts implemented by the suppliers, and thus the quality of the final product. This makes applying a generic DCS challenging for the OEM. Instead, it typically prompts the need to adapt a DCS to the specific development context.

While DCS adaptations have previously been identified as necessary (Freimut, 2001), they bring new challenges that are currently not sufficiently addressed. One such challenge involves balancing generalizability and specificity of the classification data. More concretely, in order to use defect classification data to demonstrate effects of new or modified practices, and to be able to generalize such findings to a context beyond the particular organization studied, the classification data needs to be equally generalizable. By adapting the DCS to a particular development context, it may risk inhibiting such generalization. The IEEE Std. 1044 addresses this by describing a standard structure for DCS, and by establishing mandatory elements in order to be compliant. There is, however, a lack of published experience reports with regard to IEEE Std. 1044. In addition, while there have been reports on adaptations to DCSs, there is a lack of established methods for evaluating the adapted schemes.

In this thesis, these challenges are addressed by first, in Part II, demonstrating through an industrial case study the need for data to supplement expert opinion. We then provide in Part III an in-depth description of a DCS adapted from the IEEE Std. 1044 to the development of automotive safety software. In Part IV, we report on the evaluation of the adapted DCS by first describing the results of an industrial case study assessing its practical viability. Additionally, we describe, in Part IV, a comprehensive method for evaluating DCSs. In Part V of this thesis, we

[1] Original Equipment Manufacturer

elaborate of future research direction with respect to DCSs. In the final part, we summarize and conclude this thesis.

The remainder of this introduction is structured as follows. Chapter 2 presents background material relevant to this thesis; chapter 3 summarizes the research goal and, the specific research questions addressed and finally; chapter 4 elaborates on the research methodology applied in the thesis.

2 AUTOMOTIVE SOFTWARE DEVELOPMENT AND DEFECT MANAGEMENT

Effective defect management is especially important in large, complex and dynamic organizations. In large organization there are usually many different roles involved in product development, e.g. managers, analysts, architects, designers, developers and testers. As an organization's size increases so does the number of communication paths within and between development teams (Pareto et al., 2012), adding to the complexity of the development organization (further discussed in chapter 9.3). In dynamic organizations, developers may, once their particular development phase has concluded, be reassigned new projects or different roles within the projects. Thus, the knowledge and experience from prior projects and project phases is fleeting unless made explicit. It can therefore be challenging to systematically identify development issues, as the required knowledge may not be readily available.

For this thesis, the development of software-intensive automotive active safety features at Volvo Car Corporation (VCC) has been studied. The organization developing the active safety features is large in that there are several hundred engineers involved. The organization is dynamic in that engineers are regularly reassigned between projects and project roles according to project needs and their specific expertise. For instance, as the development of a vehicle model may span several years, the engineers involved in specification will have been reassigned new projects by the time the software is implemented. The complexity of automotive software and its development comes to a considerable extent from the distributed nature of both the products (characterized in chapter 2.1) and the development process (described in chapter 2.2). An additional source of product complexity is the environment in which the active safety features are operating. Specifically, the features are designed to operate on a wide variety of road conditions around the world, and the consequences of malfunction can be severe—the amount of potential test cases the features must be put through is enormous. It is therefore important to be able to evaluate the efficiency of the various test methods—for instance, to evaluate whether the tests methods detect the intended types of defects (see chapter 8.5.2).

Effective defect management provide valuable insights that can be used to address challenges met in large, complex and dynamic organizations. Figure

1 shows a simplified view of the software development process at VCC where defect management is an on-going activity throughout projects and after release (chapter 2.2 provides a more detailed description of the figure). Through the process, the defects are typically detected by a variety of methods and roles, and documented in a format that is suitable for that particular context—often with the sole purpose of facilitating its resolution. In order to enable efficient quantitative analyses, for instance to detect systematic development issues, structured defect documentation is needed (as indicated in Figure 3 on page 10). For this thesis, defect classification as an approach to provide structured defect documentation is examined and applied to the development of automotive safety software.

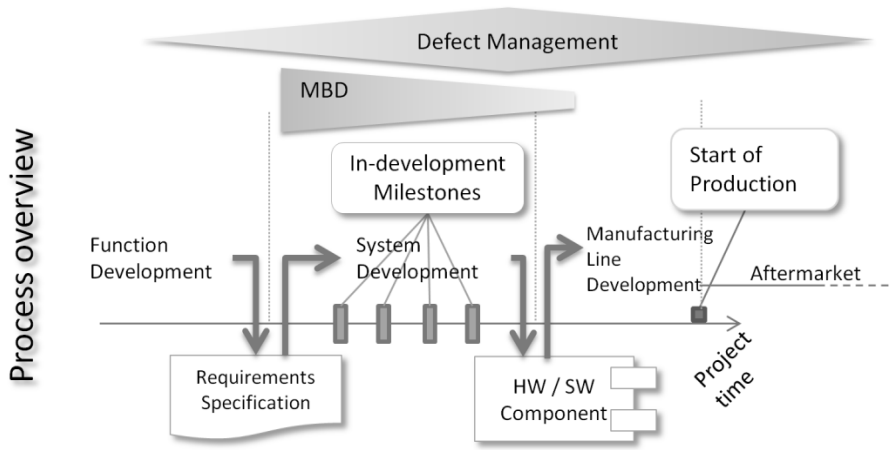

Figure 1. A simplified overview of the software development process at VCC

The remainder of this chapter provides background and context for the thesis by first describing an example active safety feature. The description aims to illustrate the complexity of the products developed, and how that complexity contributes to the need for efficient defect management (for a more in-depth description of automotive software development, see e.g. Eklund (2013, chap. 5) and Hristu, Varsakelis and Levin (2005, pp. 741–765)). In section 2.2, the development process is described in more detail. The section provides a specific focus on the defect discovery procedures and further emphasises the need for structured defect documentation. In section 2.3, a more general discussion on defects, defect classification and its relation to relevant standards is provided. Finally, as a model-driven development

(illustrated by the MDB block in Figure 1) is considered a promising approach to improving efficiency and effectiveness of software development at VCC, section 2.4 provides a more general description of software models and model-driven development (chapter 5 provides a more specific description of the use of software models at VCC).

2.1 AUTOMOTIVE ACTIVE SAFETY FEATURES

Automotive active safety features typically aim to assist the driver in preventing, or mitigating the effects of, various types of accidents. One example of such a safety feature is forward collision warning (FCW), which monitors objects in front of the vehicle using sensors (e.g. a camera and a radar/lidar). If there is a risk of collision, the feature issues the driver a visual/audible warning. Active safety features, such as the FCW, executes on a computer system deployed in the vehicle.

The computer system in a modern car is a distributed computer platform; there may be 30 to over 100 different computers (Electronic Control Units, or ECUs) in the car, connected with a number of communication busses. Figure 2 shows an example vehicle architecture, where ECUs are represented with rectangles connected with communication busses (in the example, two CAN[2]-busses, and one MOST[3]-bus). Running on each ECU are a number of programs (referred to as Software Components, or SWCs), where each SWC has a specific responsibility.

Often, a feature is realized by a number of cooperating SWCs. As a highly simplified example[4], the FCW feature may be realized by SWCs running on three different ECUs:

- An SWC running on the ECU labelled FSM (Forward Sensing Module), connected to a front-facing camera and a radar, that identifies and tracks objects in front of the vehicle;
- An SWC running on the ECU labelled CEM (Central Electronic Module) that assesses a threat level of the objects identified. The CEM does this by periodically receiving information about the objects in

[2] Controller Area Network (CAN) is a low-speed communication protocol specifically designed for and extensively used in the automotive industry.

[3] Media Oriented Systems Transport (MOST) is a high-speed communication protocol for multimedia and infotainment networking, intended for the automotive industry (http://www.mostcooperation.com).

[4] Note that the example is intended as an illustration of a possible system deployment, rather than an actual design. The ECU abbreviation spelled out in the text may therefore not correspond to the actual Volvo XC90 network topology.

front of the vehicle (from the SWC deployed on the FSM), and of the current vehicle's speed and driving direction (which it receives from the Engine Control Module labelled ECM), and finally;

– An SWC running on ECU labelled DIM (Driver Information Module) presents the driver with audible/visual warnings if there is a risk of collision. The warning is triggered by a signal sent from the SWC deployed on the CEM.

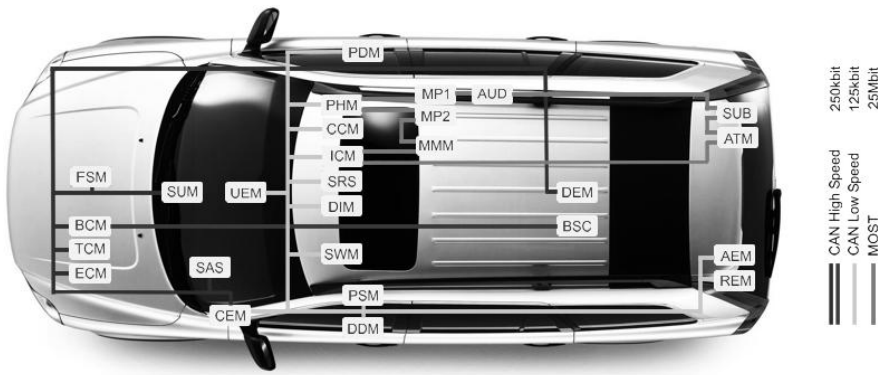

Figure 2. The figure illustrates an example vehicle architecture. ECUs are represented as rectangles, and communication busses as lines connecting the rectngles (the image, showing a modified illustration of the Volvo XC90 network topology, is adapted from Eklund (Eklund 2013))

Typically, the SWCs that cooperate in realizing a feature are developed by one team. There are, however, often cases where an SWC provides a service to multiple features; for instance, the SWC deployed on the FSM, in the example above, may also deliver information about objects in front of the vehicle to the parking assistant feature, deployed on the ECU labelled PSM (Parking Support Module). Also shown by the example above, the SWC deployed on the CEM depends on functionality deployed on the ECU labelled ECM—thus not part of the FCW feature. This example illustrates that the design and development of automotive safety features are distributed, which contributes to making defect management challenging; for instance, emphasizing the importance of a defect management approach that facilitates analyses of defects across the entire computer platform.

In the cases where an SWC provides a service for external features, a communication interface is defined for the service as a set of signals. A signal

is assigned to be sent over a specific communication bus (in this example on one of the high-speed CAN busses), has a data type and is sent at a specified interval. The communication interfaces typically need to be specified early in the project (discussed in chapter 5). Among the reasons is that the communication interfaces have an impact on the number and the types of communication busses that are needed. The choice of busses, in turn, has an impact on the physical design of the vehicle, which requires such information early—the physical design affects the vehicle's structural integrity and crash resilience, which is a costly process to ensure. The validation of the communication interfaces, however, is typically done late in the project, as the developed components become mature enough to be properly tested. Although there are state-of-the-art approaches to achieve earlier validation (e.g. models-in-the-loop and continuous integration (Giese et al., 2011)), such approaches are typically expensive, or require substantial process changes. Thus, deciding to apply such approaches typically need justification. A system of proper measurements can support justifying such decisions, and also evaluating the eventual effect of the improved practices.

2.2 AUTOMOTIVE SOFTWARE DEVELOPMENT PROCESS

The studies included in this thesis have been conducted within the active safety department at Volvo Car Corporation (VCC). The description of the development process at that department, which is provided below, is also presented in chapters 5.4.1 and 6.2.3. In each of these chapters, the process is presented from the perspective, and emphasizing the details, relevant to the particular study presented in that chapter.

In Figure 1, an overview of the development process is presented. On a highly abstract level, the in-house process of developing automotive software systems can be divided into three main phases as shown in Figure 1:

- *Function development*: The objectives of the function development phase include developing requirements specifications and high-level designs for all functions in the vehicle model. Specifically, the high-level design decomposes each function into a number of SWCs, and on which ECU each is to be deployed. In addition, a requirements specification for each SWC is developed. The design and requirements specification is provided as input to the following phase;
- *System development*: The objective of the system development phase is to develop each ECU. This includes the physical component itself, and the various SWCs deployed on it. In this phase, a development team is assigned a number of SWCs to develop. The input to each team is the

set of requirements that apply to the SWCs. The output of the phase is a set of ECUs with SWCs installed on them;

– *Manufacturing line development*: The objective of the final phase is to develop the factory assembly line in which the cars are mass-produced. This development phase is relevant to the software development process, as certain components in the car are calibrated as part of the manufacturing process. Such calibrations are often realized by the software.

The main focus in this thesis is on the *System development phase*. In the system development phase, each team develops a number of SWCs in accordance with the requirements specification. Normally, each SWC is implemented as an executable model in a tool such as Simulink. From these models program code, such as C, can be generated. This is illustrated in Figure 1 by the block labelled MBD (model-based development).

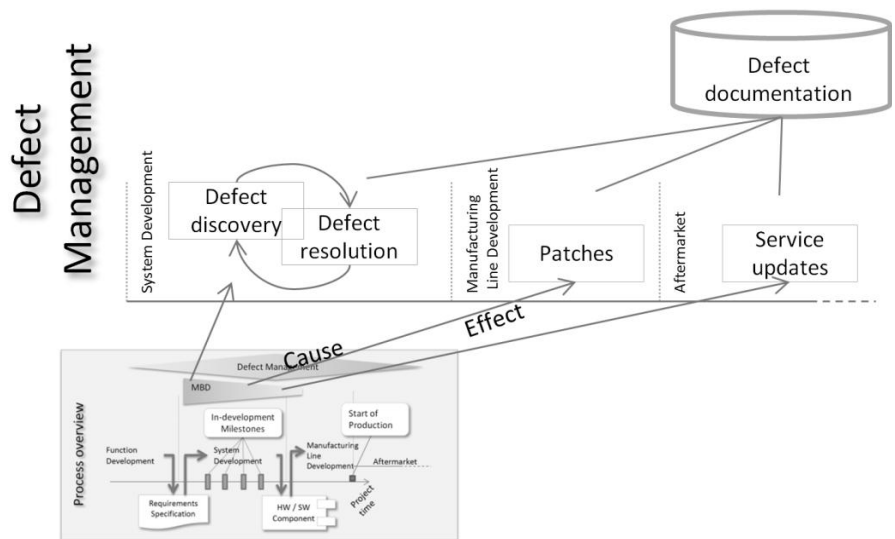

Figure 3. A simplified view of a defect management process

Although, the main focus in this thesis is on the system development phase, defects discovered during this and the subsequent phase are of interest. The reason is illustrated by the arrows labelled '*Cause'* and '*Effect'* in Figure 3. The arrows illustrate that defects introduced in system development may be detected in later project phases—this also applies to defects introduced in

other artefacts, such as requirements and design. As indicated by the arrows, defects may remain in the system until triggered by specific tests. Identifying the effectiveness of the test activities, and which development artefacts are error-prone, in a distributed development environment is a challenge. Moreover, while expert knowledge can provide information about these aspects, such knowledge is typically tacit. As development often span over several years that knowledge may not be readily available when needed. This provides further justification for a defect management approach (illustrated in Figure 3) that makes the tacit knowledge explicit, and that facilitates efficient analyses.

In addition, the distribution of development among in-house teams, the system development process typically involves a number of external suppliers in addition to engineers at the vehicle manufacturing company (OEM). Whereas the engineers at the OEM develop an executable model for each SWC, the suppliers are typically responsible for delivering the ECUs (the physical component) with SWCs installed. Specifically, the supplier generates C-code from the executable model, then optimizes the code and compiles it for the particular type of ECU on which it is to be deployed. The result is a binary SWC that is delivered to the OEM. While in most cases, the suppliers are responsible for the binary SWCs along with developing the ECUs themselves, in others cases, they are responsible only for the binary SWCs. This distribution of responsibility, both between process phases at the OEM, and between the OEM and the suppliers, makes root cause analyses a challenge. Without proper measurements, it may be difficult to identify the process interfaces—e.g. the artefacts delivered between project phases or between the OEM and the suppliers—that are problematic.

Although the overarching process in the automotive domain is typically described as a waterfall—often visualized by the standard V-model—the systems development phase is in fact iterative. Throughout the phase, there are a number of milestones, each intended to test the functions at an increasing level of maturity. In early stages of the development phase, the majority of tests are so called bench-tests, in which a sub-set of the vehicle's electrical system is simulated in a lab environment. In the bench-tests, ECUs may be simulated with regular computers or various types of specialized hardware (e.g. dSpace-modules). As the project progresses, the bench-test setup will include pre-production versions of the final ECUs, and later complete test vehicles are built to test the functions; running particular scenarios on dedicated test tracks, and field tests (called expeditions), where the vehicles are driven on regular roads.

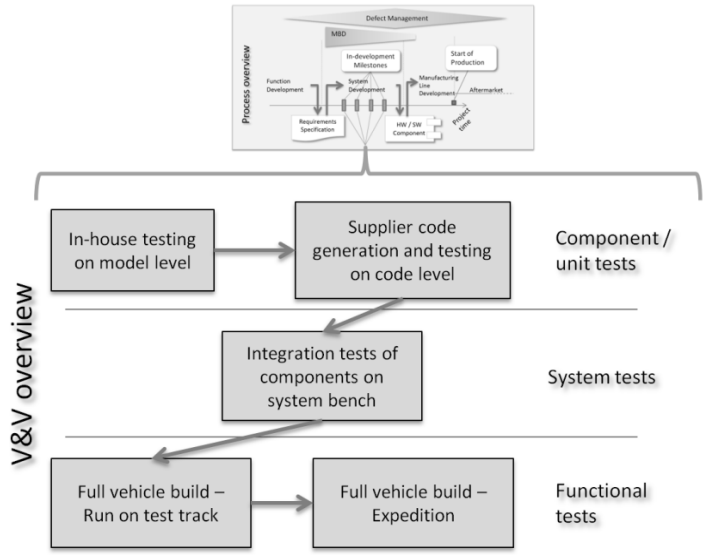

Figure 4. An overview of the typical validation and verification procedures done for each milestone in the system development phase

More specifically, for each milestone, the development teams prepare a software delivery consisting of executable versions of the SWCs they have been tasked to develop. In preparation for each delivery, the development teams typically follow the testing procedure illustrated in Figure 4:

- The executable models are tested on unit level by the development team at the OEM;
- The binary components, generated from the executable models and optimized, are tested on unit level by the supplier. Any defects found are reported back to the OEM;
- Initial integration tests are run on bench by the OEM. In these tests, a number of SWCs are cooperating to realize functionality. The focus of these tests is to ensure that communication and synchronization between SWCs are working as intended. An often used test method (re-sim) uses recorded data in place of sensors, for instance recorded video instead of a real camera;
- In the final type of test (Functional test), the complete function is tested. In this type of test, a full vehicle is built with all the SWCs installed. In the first type of functional test, specific scenarios are run on a dedicated test track; for instance, the frontal collision warning

system may be tested by driving the vehicle into balloon cars (inflatable objects). In later phases of development, vehicles are tested on regular roads (expeditions). The purpose of this type of test is to ensure that the functions behave as intended in their real environment; for instance, that the collision warning system does not give false warnings.

While the test procedure, as shown in Figure 4, is typically followed for each milestone in the system development phase, the focus is gradually shifted towards functional test; i.e. initially mainly component and system tests are run, while in later project phases the focus is on specific scenarios on the test track, and on expeditions.

Each of the types of tests shown in Figure 4 is associated with a cost—it is, for instance, generally cheaper to detect a defect in a unit test than in a functional test. It is of interest to the organization to evaluate that the test types are effective in detecting the types of defects they were designed to detect. Assessing the effectiveness, however, requires systematic data collection and analysis.

2.3 DEFECTS

As the concepts of a defect and of defect classification are central to this thesis, this section is dedicated to their definition and a discussion on their relation to relevant standards.

2.3.1 Definitions

The definition of the term *defect* is in this thesis taken from IEEE Standard for Classification of Software Anomalies (IEEE Std. 1044-1993 (IEEE, 1993)). The standard was originally issued in 1993 with a subsequent revision issued in 2009 (IEEE, 2009). Interestingly, the two versions of the standard use different definitions of the entity intended for classification (both shown in Table 1). Whereas the 1993 version defines the classified entity as *anomaly*, the 2009 version instead uses the term *defect*. The definition of the term *anomaly* (shown in Table 1) is an expansion of the more limiting definition given in ANSI/IEEE 610.12-1990 (IEEE, 1990), also including "*deviations from the user's perceptions or experiences*". In the 2009 revision, the more specific term *defect* is used instead. The reason—as stated in the introduction—was that the term *anomaly* was considered too broad to lend itself to efficient communication.

Table 1 The definitions of the classified entity in the two revisions of the IEEE Standard Classification for Software Anomalies (IEEE Std. 1044)

Source	*Definition*
IEEE 1044-1993	***Term: Anomaly*** Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc. or from someone's perceptions or experiences. Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.
IEEE 1044-2009	***Term: Defect*** An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.

In this thesis the term *defect* is used, but with the broader definition of *anomaly* from IEEE Std. 1044-1993. The justification for using that definition is that experience from our case company has shown that there are occasionally issues (as defined by (ISO/IEC/IEEE, 2010)) in which the design or implementation of functionality comply with the specification, but would not meet users' expectations. As an illustrative example, in the initial phases of a vehicle project (referred to as the *Strategic phase* in chapter 5.4 on page 38) high-level requirements on the expected behaviour of each feature are specified. Such high-level requirements may often be vaguely expressed; e.g. "*feature shall be world leader in the premium car segment*" or "*feature shall interact with the driver in an unobtrusive and natural manner*". Even though subsequent refinements are considered to comply, it may be found when the feature is finally implemented that it fails user expectations—i.e. when the functionality is successfully verified but fails validation. Such cases would not be included in the more restrictive definition provided by the IEEE Std. 1044-2009.

2.3.2 Preventing Defects

Proper defect management is important in any software development domain. In the automotive domain, the importance is further emphasized by the safety aspects. For instance, a trend in recent years has been towards features which are able to make increasingly autonomous interventions, such as braking or

steering. Malfunctioning software may therefore cause severe injuries to the vehicle occupants, and to individuals in the vehicle's surroundings. Standards, such as ISO/IEC 26262 (further discussed in section 2.3.4) aims to assist in analysing and thereby assuring safety of such features.

Still, there are examples of malfunctioning automotive software. For instance, in 2010 there were reports concerning the new Toyota Prius model, where brake performance was decreased under specific circumstances[5]. While the specific fault(s) underlying the failure was quickly identified by Toyota and addressed in a software update, a more interesting question to the company would be how the fault was allowed to slip through the verification process. By identifying how this could happen it would also allow preventing similar incidents from happening in the future. Root-cause analysis (RCA) is one approach to providing such in-depth investigations. Equipped with knowledge of the root-cause, a plan of action to avoid similar future incidents can be set up. RCA, however, has been shown to require substantial effort (Chillarege and Inc, 2006) and may be feasible to apply only in exceptional cases. RCA is typically reactive in that it is applied after the fact.

An additional approach to learning from experience is post-mortem reviews (Dingsøyr, 2005), which is indented as a continuous activity for process improvement. At the conclusion of a project, an appointed team is tasked to critically review the project and provide lessons learnt—positive as well as negative experiences. While RCA can be considered a point-effort to be applied in specific cases, post-mortem reviews are continuous in that they are intended as an integral part of the project life-cycle.

Post-mortem reviews are, as with RCA, reactive in that they are performed after the fact. To enable pro-active defect management—i.e. to prevent defects—continuous in-process activities are required that provide information about likely sources of defects; techniques that, for instance, based on change patterns in source code can indicate increased risk of specific types of defects, thereby indicate which types of tests need to be performed. One of the required components in such a technique is structured defect documentation, and defect classification schemes are one promising approach.

2.3.3 Defect Classification Schemes

In practice, the purpose of a defect report is often limited to facilitating the resolution of the defect. In particular, defect reports are often free text

[5] The investigation report (action number PE10006) by NHTSA is available from: http://www-odi.nhtsa.dot.gov/owners/SearchNHTSAID

(Wagner, 2008)—the reports do therefore not easily lend themselves to efficient quantitative analyses. To address this problem, various defect classification schemes have been proposed. Such classification schemes contribute to comparability of defect metrics between projects, and between companies (Chillarege et al., 1992).

A useful classification scheme comprises a set of attributes, where each attribute captures a specific aspect of the defect—e.g. how the defect was detected, its severity and type. Moreover, for each attribute, the schemes typically provide a set of values that can be chosen from; this contributes to the efficiency as well as to the reliability of the classification.

The most commonly referred (Freimut, 2001) classification schemes in literature are ODC (Orthogonal Defect Classification) (Chillarege et al., 1992), the HP scheme (Grady, 1992) and the IEEE Std. 1044 (IEEE, 2009, 1993). Common among these established classification schemes is that they have focus on source code, and on low level attributes related to the implementation. In addition to requiring access and detailed knowledge of the implementation, it also risks missing important aspects of defects in artefacts that does not have direct manifestations in the source code; e.g., quality of specifications and missing or inadequate test cases.

In order to be useful, defects should be classified in-process, i.e. aspects of the defects should be documented according to the classification scheme as soon as the relevant information is revealed. One way is to incorporate the DCS into the organization's issue-tracking system. The DCS will in such case serve as a defect report template, prescribing what aspects of the defect must be documented as part of the defect life-cycle (e.g. detection, analysis, and resolution). Performing the classification in-process—when the details of the defect are still in fresh memory—increases the likelihood of correct classification, and will likely minimize the time required (indications of which are reported in chapters 8.5.1 on page 132 and 8.6.1 on page 144).

In this thesis a DCS based on and compliant with IEEE Std. 1044, and adapted to the development of automotive active safety software is described. The resulting DCS is named Light-weight Defect Classification scheme (LiDeC).

2.3.4 Standards

Applying a DCS for characterizing and documenting defects is important to organizations that want to comply with standards. Complying with standards contribute to repeatable defect management and to the consistency of the defect data over time. The standards, in addition to IEEE Std. 1044 as

discussed in the previous section, that are relevant to the DCS described in this thesis relate primarily to various aspects of product and process quality.

ISO/IEC 9126 (ISO/IEC, 2001) defines a quality model that is relevant to the *Effect* attribute in LiDeC (see Appendix A, page 187). The ISO/IEC 9126 is a standard for evaluating software quality. Specifically, it describes a quality model as a set of characteristics (i.e. *quality attributes* as defined by (ISO/IEC/IEEE, 2010)). The characteristics are *Functionality*, *Reliability*, *Usability*, *Efficiency*, *Maintainability*, and *Portability*. Each characteristic is further divided into more fine-grained sub-characteristics. Such sub-characteristics are useful to be able to classify the *Effect* attribute with a higher granularity—thus, provide an extensibility mechanism for the attributes in the DCS. ISO/IEC 9126, furthermore, defines three types of metrics: internal metrics, external metrics and quality in use metrics. In this thesis, mainly external and quality in use metrics are relevant, as the main test activities at the case company involve code execution.–There are a number of additional standards related to software quality assurance (SQA). While these standards specify aspects of various best software development practices, the specifics regarding their implementation are left for the individual development organization to select. Therefore, DCS are relevant as a concrete technique to implement concepts specified by these standards.

The relevant SQA standards can be classified as quality management and project process standards (Galin, 2004, p. 473). The quality management standards focus on the SQA infrastructure while leaving the choice of particular methods and tools to the organization—i.e. they focus on *what* to achieve by the SQA process, rather than on *how* that is to be achieved. Prominent quality management standards include ISO 9000-3 (ISO, 2005, pt. 3), CMMI (CMMI Product Team, 2010a, 2010b) and ISO/IEC 15504 (also known as SPICE, and with its more specialized version Automotive-SPICE) (ISO/IEC, 2004; The SPICE User Group, 2011). Project process standards on the other hand focus on how a project is to be implemented. The standards define which steps are to be taken, and what work products are to be developed. Prominent project process standards include ISO/IEC 12207 and IEEE Std 1012-1998.

Quality management standards generally serve at least two purposes. Firstly, to enable assessing the maturity of an organization's SQA infrastructure; this typically involves certification by an external accredited body. Such certification allows for instance OEMs to assess a supplier's SQA process, and thereby provides an indication of an organizations ability to deliver high quality products. Secondly, the quality management standards can be used to guide an organization in improving their SQA system.

A prominent example of a quality management standard is ISO 9000-3, which define a set of requirements pertaining to all aspects of an organization's software development process. In order for an organization to be certified, it must demonstrate its compliance to each of the requirements. Typically the certification process is conducted in collaboration with an authorized certification body.

The capability maturity model integration (CMMI) is, similarly to ISO 9000-3, a model used for assessing the capability of a software development organization. Whereas according to ISO 9000-3 an organization is either certified or not, CMMI defines five levels of process area maturity. As with ISO 9000-3, the five levels of maturity defined by CMMI allow both external assessment of an organizations capability, and internal guidance on how to reach the next level of capability. The particular focus of CMMI is on management methods based on quantitative approaches. This is reflected, for instance in that at the higher levels maturity, an organization is required to utilize product and process metrics to control performance and identify improvement needs and opportunities (Galin, 2004, p. 504). The goals and structure of the ISO/IEC 15504 standard is similar to CMMI in that it defines a number of maturity levels for a set of key process areas. The similarity between CMMI and ISO/IEC 15504 has been showed through mappings between the two standards (e.g. (Peldzius and Ragaisis, 2011; Rout and Tuffley, 2007)). Such mappings are justified by the desire of organizations that have been certified according to one of the two standards, to also become certified according to the other—indicating that the two standards are overlapping.

The quality management standards typically focus on what a development organization shall be capable of without specifying how to achieve that capability. This is an intentional approach as the specific methods and tools most appropriate to achieve the capabilities may be different between organizations (for instance, depending on product domain). The methods and tools chosen, however, need to contribute to the capabilities defined by the standards. As such, defect classification schemes are a method contributing to the achievement of level 4 in CMMI and ISO/IEC 15504 (specifically *Measurement*).

Project process standards specify, in contrast to quality management standards, how to perform a process. IEEE/EIA 12207 and IEEE 1012 are two such standards. Even though these standards describe which steps are to be taken and what work products are to be developed as part of the process, they still leave the concrete techniques (e.g. methods and tools) open to the

organizations. The concept of a DCS is, in this context, relevant as one technique to meet specific requirements posed by the standards.

The main purpose of the project process standards is to define a common framework of best practices. These best practices are in the form of activities and sub-activities that should be performed as part of a project. While IEEE/EIA 12207 covers the entire project life-cycle, and IEEE 1012 focuses specifically on verification and validation activities (V&V), the two standards are closely related —Annex A in IEEE 1012 provide a mapping to IEEE/EIA 12207 tasks.

IEEE/EIA 12207 defines, using a hierarchal structure, processes (e.g. *acquisition*, *supply*, and *development*) and for each process a set of activities that shall be performed; for the *supply* process, activities may include *initiation, preparation of response, planning, execution and control* (Galin, 2004, pp. 530–531). For each activity, furthermore, the standard defines a set of tasks in the form of requirements; the execution and control activity may, for example, include the task (Galin, 2004, p. 531):

> *Supplier shall monitor and control the progress of development and quality of software products, including problem identification, analysis and resolution*

While the project process standards aim to define processes more concretely than the quality management standards, they still leave it open to organizations how to implement their processes. More specifically, the standards define requirements that need to be met in order to be compliant—i.e. it is up to the organization to choose the particular methods and tools suitable for each activity. Defect classification schemes may therefore serve as a technique fulfilling certain task requirements. For instance, in the task requirement above, the supplier is to monitor the quality of software products, but the implementation of that monitoring activity is not specified. Defect classification can contribute with data regarding one aspect of the quality of the software.

The ISO/IEC 26262 (ISO/DIS, 2011) standard is intended to support the development of safety-critical automotive features. Specifically, the standard defines a set of best practices in specifying, designing, implementing and verifying safety relevant aspects of automotive features. While the standard defines best practices for software testing at various levels (clauses 9-11 in ISO/IEC 26262-6 (ISO/DIS, 2011, pt. 6)), it does not specify details on how to document defects. DCS is one technique for documenting discovered defects that comply with the standard, while capturing data on additional aspects of the defects.

In essence, defect classification schemes are a technique for extracting measurements from defects. As such, the standard ISO/IEC 15939, *Systems and software engineering—Measurement process* (ISO/IEC 15939, 2007) is of relevance to this thesis. ISO/IEC 15939 defines activities and tasks necessary for defining and applying a measurement process in a software development organization. Of specific interest for this thesis is the measurements information model in Annex A (ISO/IEC 15939, 2007). The information model defines key concepts related to measurements, and their relationships. In this thesis, chapter 5 describes a case study which came to reveal *information needs* (as defined by the standard) that were not satisfied. The act of classifying defects corresponds to *measurement method* in the standard, and the classification data to *base measures*. Furthermore, the design of LiDeC (as reported in chapter 6.3) followed the process defined by the standard. While the additional concepts shown in the standard's information model are important to form a practical measurement product, it is outside the scope of this thesis—although, chapter 9 outlines a research roadmap that aims to incorporate the remaining concepts of the information model.

2.4 MODELLING

Improving development efficiency and product quality are typically on-going efforts in software development organizations. However, identifying improvement opportunities and evaluating the effects of undertaken improvement initiatives in a complex development organization is challenging. While defect classification data is one means of facilitating identification and evaluation of improvement initiatives, chapter 5 describes a case study exploring an alternative way.

In the case study, it was presumed that adopting a model-driven development approach (MDD) would improve development efficiency and product quality. In the study, the centrality of software models in a non-model-driven development organization was examined. More specifically, the study contrasted modelled and non-modelled development artefacts from the perspective of their respective perceived cost and benefit. The intended outcome was to be able to better identify which development activities would benefit most from applying an MDD approach. To provide chapter 5 with context, a more general definition of software models and model-driven development is given here.

Representing development artefacts as models is common practice in many software development organizations. In the development of active safety

software at VCC, one of the most common type of model is implementation model (e.g. Simulink models (Mathworks, 2010)). Representing a development artefact as a model (such as Simulink model in place of source code) has the purpose of providing an abstraction for a particular purpose. Mellor et al. (Mellor et al., 2003, p. 15) provide the following definition of a model:

> *A model is a coherent set of formal elements describing something (for example a system, bank, phone, or train) built for some purpose that is amendable to a particular form of analysis […]*

The principle of MDD (Kent, 2002; Mellor et al., 2003) is that models should be the primary development artefact, i.e. work should be done on the models, and other artefacts (such as document or code) should be (semi-)automatically generated from the models. Ideally, the development should comprise a chain of model transformations, from requirements to source code. In each development step, analyses are conducted; information is refined and added to the models, allowing it to be transformed to the next and more concrete model eventually resulting in an executable artefact.

Frequently cited reasons for improved development efficiency with MDD include: a) as each type of model is designed for a particular kind of analysis, that analysis should be more efficient, and b) as multiple types of artefacts can be generated from the models is should reduce the amount of manual development work.

Improved product quality is typically attributed model transformations, which requires less manual work than code-centric development. For instance, as source code can be generated from the models, it significantly reduces the amount of manually written code, which is often error-prone. This means that verification and validation can be done on model level, enabling more direct validation of domain specific concepts, rather than obscured by specific programming language constructs.

While MDD can improve aspects of software development, it also requires changes to the development organization. For instance, it has been shown that the focus of the organization needs to be shifted from writing source code, to developing meta-models and model transformations. Making such a transition in a large or complex development organization is challenging and risky. In order to mitigate such risks, a more targeted adoption may be effective (Selic, 2003)—i.e. to introduce modelling in the parts of the development process that present the best opportunity for improvement. In addition to methods for identifying such process parts, methods evaluating

the impact of improvement initiatives are required. In particular, a broad perspective needs to be considered in order to assess whether the initiatives accomplished the intended improvements, and not merely local optimizations while causing issues in other parts of the development process. By analysing classified defect data, such effects can be evaluated.

3 RESEARCH FOCUS AND THESIS STRUCTURE

The main research question in this thesis is summarized as:

> *How to improve defect management in automotive software development using structured defect documentation?*

The majority of this thesis is a continuation of our previous research (Mellegård, 2010). In that research, the goal was to examine how model-driven development methods could improve automotive software development practices. Among the results of that research were the realisation that, while project staff often have strong opinions of what development practices need improvement, a significant challenge is to provide evidence that those improvements actually targets the relevant development issues. Consequently, the main contribution of this thesis is the proposal and evaluation of a defect classification scheme that can be used as a source of such evidence—thus, the aim of the main research question is to improve defect management by extending the usefulness of defect documentation beyond merely facilitating the resolution of the defects.

3.1 RESEARCH QUESTIONS

The main research question was broken down into four separate research questions; each question is addressed in a separate part of this thesis.

3.1.1 Characterizing Automotive Software Development

The first research question, addressed mainly by our previous research (Mellegård, 2010), aim at characterizing the development of automotive software from a modelling perspective. The research question was:

> ***RQ 1*** *How important and effort intensive are software models in automotive software development, and what are the opportunities for improving the utilization of models?*

This research question is addressed in Chapter 5 in Part II by providing an overview of the current way of working with a focus on the use of software models. The aim of the chapter was to investigate how the use of software models could be made more efficient by characterizing the current way of working. More specifically, the chapter describes a study contrasting the

perceived cost and importance of the various model and non-model related development artefacts.

3.1.2 Adapting a Defect Classification Scheme

In addressing research question 1, we found that although there were many expert opinions regarding which development practices were in need of improvement, there was a lack of verifiable evidence. Consequently, the research focus shifted towards investigating how to obtain such evidence. In Chapter 6 of Part III, the following research question is addressed:

RQ 2 *How to improve efficiency of applying IEEE Std. 1044 in model-based automotive software development?*

The aim of research question 2 was to examine how to adapt the standard for classification scheme (IEEE 1044) to automotive software development, while maintaining compliance with the standard. As a result, LiDeC (Light-weight Defect Classification scheme) was developed, as is reported in-depth in Part III.

3.1.3 Evaluating a Defect Classification Scheme

The third research question, addressed in Part IV, aims at providing an evaluation of LiDeC. Part IV addresses the two research questions:

RQ 3.1 *How feasible is LiDeC for model-based automotive software development?*

RQ 3.2 *How to evaluate a defect classification scheme?*

The aims of the research question addressed in Part IV were two-fold; firstly, to evaluate the practical viability of LiDeC, as reported in chapter 7. Secondly, to examine, describe and reflect on a comprehensive method for evaluating defect classification schemes, as reported in chapter 8.

3.1.4 Future Directions

The final research question aimed at examining how the concept of DCS can be further developed. Specifically, chapter 9 of Part V addresses the research question:

RQ 4 *What are the future research directions for improving defect classification?*

By examining the published research up to this point, three main areas in need of further research are outlined. Addressing these research areas would further the applicability of defect classification schemes as an industrial as well as academic tool for data collection.

3.1.5 Mapping of Research Questions to Chapter Contributions

Research question 1 is addressed in Chapter 5 by investigating the following more specific questions (prefixed with *M* for Modelling):

M1.1 *Which models are used?*

M1.2 *Which modelling notations are used?*

M1.3 *What information is conveyed by the models?*

M1.4 *How much automation is used in the development of models?*

M1.5 *Which documents are created from the information contained in the models?*

M2.1 *What is the relative importance of the model-related artefacts in the software development process?*

M2.2 *What is the relative cost (in terms of effort) for developing model-related artefacts in the software development process?*

Research question 2 is addressed in chapter 6 by investigating the following more specific questions (prefixed with *L* for LiDeC):

L1 *In an automotive safety feature development context where source-code is often not available, how can a standard defect classification scheme be suitably adapted?*

L2 *As defect classification may often be considered an administrative task, how can the adaptation of a standard defect classification scheme be done to minimize required learning and classification time?*

Research question 3.1 is addressed in chapter 7. While the more specific questions L1 and L2 in chapter 6 focused on developing an adapted defect classification scheme, chapter 7 addresses two similar questions but with the focus on evaluating the classification scheme in an industrial context. The more specific research questions investigated in chapter 7 are (prefixed with *E* for evaluation):

E1 *How can the IEEE Std. 1044 be adapted to a software development context with limited access to source-code?*

E2 *How can the IEEE Std. 1044 be adapted to minimize its process foot-print in terms of required learning and classification time?*

Research questions 3.2 and 4 are addressed in chapters 8 and 9 respectively. The chapters do not pose more specific research questions.

3.2 RESEARCH APPROACH AND THESIS STRUCTURE

The overarching research approach employed in the work in this thesis followed the technology transfer model, as depicted in Figure 5 (adapted from Ivarsson (2010), originally published in Gorschek et al. (2006)). The purpose of the model is to facilitate smooth transfer of technology from academic research to industrial practice. More specifically (from Gorschek et al. (2006)):

> *"Technology transfer, and thus industry-relevant research, involves more than merely producing research results and delivering them in publications and technical reports. It demands close cooperation and collaboration between industry and academia throughout the entire research process"*

To this end, the model defines a number of steps that each, when conducted properly, contributes with both knowledge building and industrial dissemination of that knowledge. Specifically, the approach begins, as can be seen in the figure, with identifying a problem. The problem is mainly situated in an industrial context but is also of academic interest. The second step is to formulate the problem in part by studying state-of-the-art. In the third step, a candidate solution is proposed, while steps 4–6 aim at evaluating the candidate solution. Specifically, in step 4, validations in academia are conducted. Static validation, shown as step 5 in Figure 5, represent a limited scale industrial evaluation, mainly feasibility studies—for instance, to evaluate whether the solution addresses the problem, or is applicable in the specific context. Step 6, dynamic validation, is generally larger scale industrial studies, such as a pilot study, in which the solution is applied in a real industrial context. Finally, when the validations are considered successful, the technology can be released (i.e. utilized in a full industrial scale).

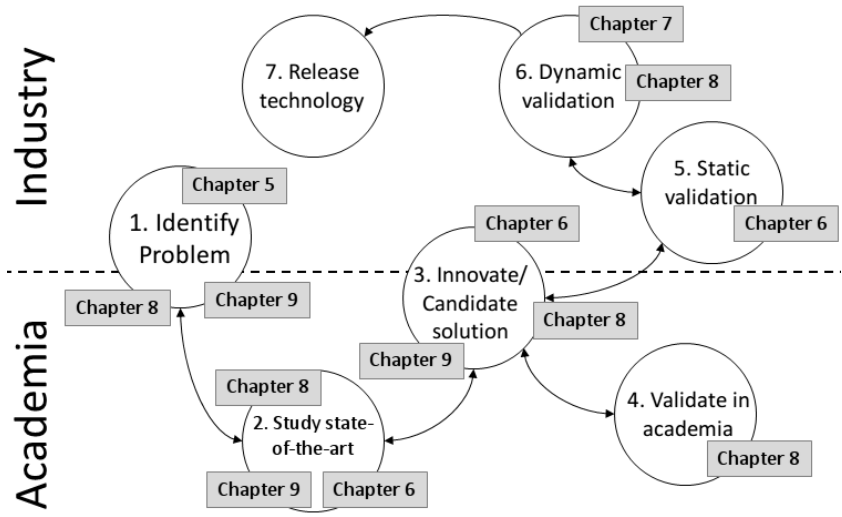

Figure 5. An overview of the research approach (adapted from (Ivarsson 2010)) in this thesis, and how the chapters of the thesis relate to the approach

The main contributions of this thesis, presented in chapters 5 to 9, add to each of the six first steps. For the first step, three separate but related problems identified and addressed in this thesis—one industrial and two academic problems.

The industrial problem is described in chapter 5. The chapter describes an industrial case study in which a more objective source of evidence regarding process improvement opportunities was found lacking. Although the main contribution of chapter 6 is towards proposing a candidate solution (step 3), it also reports on an investigation of current state-of-the-art; an investigation that led to the examination of defect classification schemes as a promising approach for collecting such objective evidence. Chapter 6, furthermore, reports on a static validation by presenting a three-stage case study in which LiDeC was developed. Chapter 8 reports on a validation of LiDeC in academia using a controlled experiment, and a dynamic validation is presented in chapters 7 and 8. The dynamic validation was conducted as an industrial pilot study in which LiDeC was applied to a set of defects from a real project.

In addressing the industrial problem identified in chapter 5, two related academic problems were identified. The first problem, described in chapter 8,

concerns that it is not clear how to validate efficiency, accuracy and reliability of adapted DCSs, even though it is recognized that a DCS need adaptation in order to be feasible. The chapter additionally presents a study of state-of-the-art, and a candidate solution, in the form of an in-depth description of a comprehensive method for evaluating a DCS.

The second academically relevant problem is identified in chapter 9 by demonstrating that there is a general need to establish defect measurement methods, and that current state-of-the-art is lacking in several respects—contributing to steps 1 and 2 in Figure 5. The chapter presents a candidate solution by proposing three main research areas that need addressing in order for DCS to become more widely adopted.

4 METHODOLOGY

Although the work presented in this thesis is academic research, it has a strong empirical focus. This is illustrated by step 1 in Figure 5, in that the main problem the thesis addresses is situated in an industrial context. Specifically, the main research goal in this thesis was to develop a feasible method for identifying improvement opportunities in automotive software development. As a consequence, two related problems mainly of academic interest were identified.

Due to this approach, a number of different research methods have been applied. The main methods and a brief validity evaluation are presented in the following two subsections.

4.1 RESEARCH METHODS

The work presented in this thesis is the result of five separate studies, presented in chapter 5 to 9. Each of the five studies had a number of specific goals, and each utilized the specific research method that was best suited for the purpose. As can be seen in Figure 6, a variety of methods has been applied—ranging from qualitative to quantitative and from empirical to analytical (Glass, 1994). Although quantitative methods collect data from a large number of cases, that data is typically shallow. More specifically, while interesting patterns can be detected in quantitative data using statistical methods, such analyses does not explain *why* specific patterns exist. In contrast, quantitative methods make deeper investigations into a small amount of cases, and aims precisely to gain understanding of underlying reasons or motivations.

Moreover, the separation between theoretical and empirical methods attempts to illustrate the method's proximity to the practice it intends to examine. In this thesis, the focus is, as can be seen in Figure 6, on empirical methods. This focus was chosen, as the goal of the work was to conduct research that was readily applicable. The analytical methods applied were literature study and review.

As each chapter discusses in more detail the methods used, the following subsection provides a more general methodological discussion.

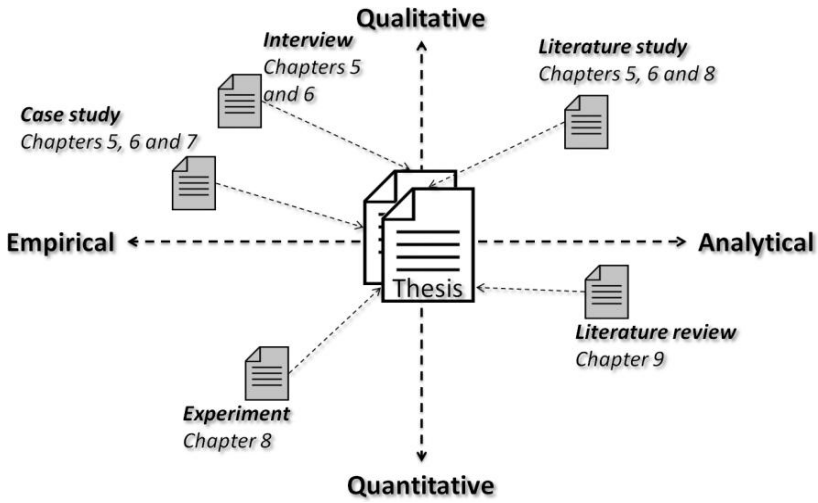

Figure 6. Thesis methodology overview

4.1.1 Case study

The main method in this thesis, because of the strong focus on empiricism, has been the case study method. Yin (Yin, 2009, p. 18) defines the case study method as:

> *"[…]an empirical inquiry that: investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident [...] and in which multiple sources of evidence is used"*

As the case-study methodology puts emphasis on the empirical study of the phenomenon in its real-life context, it was considered appropriate for addressing aspects of the first three research questions. Additionally, what was considered *phenomenon* and *context*—i.e. what parts of the development process that were relevant—was not clear, which further suggested the case-study as an appropriate method to apply.

In chapter 5, case study was applied in order to build an understanding of the current development process—partly from how that process is prescribed, but mainly how it is enacted. For that purpose, it was important to elicit qualitative evidence, in the form of document inspection and interviews, from the real-life context.

In chapters 6 and 7, the case study method was applied in order to develop LiDeC. The case study method was considered appropriate because the industrial applicability of LiDeC was central. Thus, by conducting the study in the real-life context assisted in continuously evaluating the feasibility of LiDeC (shown in Figure 5 as static validation).

While the case study method has the ability of providing nuanced results, its main weakness related to the universality of the results. Precisely because the case study method is conducted within a real-file context, it can be questioned how representative that context is to a general population—i.e. whether the results can be generalized (external validity) (Yin, 2009, p. 38). Yin (Yin, 2009, p. 116), proposes triangulation in order to minimize the threat to external validity. In this thesis, data and method triangulation has been used. Data triangulation was implemented by incorporating subjects from different projects, and in different roles. More specifically, in the case study reported in chapter 5, subjects from a number of different subprojects were interviewed, and subjects in a project management role along with developers and testers. This widens the perspective of the data collected, and reduces the risk of acquiring only one specific view. Method triangulation was done mainly by literature study, in order to confirm or refute the findings from the case study.

4.1.2 Controlled Experiment

Whereas the case study method acknowledges that—for the validity of the results—complexity and context matters, a controlled experiment aims at isolating and examining specific aspects of the object under study. In chapter 8, the aim was to examine whether LiDeC, as a classification scheme adapted to a specific context, provided additional benefits as compared to a generic classification scheme. In order to draw conclusions from such a comparison, it is necessary to eliminate all perceivable confounding factors.

While the strength of a controlled experiment is precisely that any effects seen in the results can be attributed to the treatment used, the validity of those results in a real-world context may be disputed—the confounding factors that are eliminated as part of the experiment design can be significant in a realistic setting. In addition, controlled experiments are often used to perform academic evaluations (shown as step 4 in Figure 5), thus using university students as subjects. Such academic evaluations are generally chosen because they offer low-cost and low-risk alternatives to industrial evaluations (Gorschek et al., 2006). The drawback, on the other hand is that the representativeness of the results can be questioned.

The results of the experiment, reported in chapter 8, were triangulated with the results from the industrial evaluation (using a case study method) reported in the same chapter. The results from these two evaluations offered some overlap, which provided opportunity for discussion. This triangulation strengthens both results.

4.1.3 Literature Study and Review

In addition to the two empirical methods, two related analytical methods were used; literature study and literature review. Literature study is used in chapters 5 to 8 (although the literature study in chapter 7 is included and expanded in chapter 6) to provide context and motivation prior to planning the studies. The method is qualitative and analytical. It is analytical in that it summarizes existing knowledge without attempting to provide an explicit application. Literature study is qualitative, in that a selection of publications are examined, interpreted and described in a particular context. A weakness of this method is that the selection of publications is dependent on the researcher's discretion. This weakness is to some extent mitigated by the peer-review procedure—serious inadequacies in the selection, interpretation or description of publication would likely be called into question as part of the review.

Literature review, in contrast to literature study, aims at providing quantitative data. The purpose of a literature review is to give a broader overview of a research field, by conducting targeted searches in available publication databases. A review protocol established prior to the study is used as a selection mechanism, and the resulting matches are analysed and categorized. These categories are used to draw conclusions about the current state of research within the chosen research field.

In this thesis, a small-scale literature review is reported in chapter 9. The review was conducted in order to conceptualize the use of defect classification schemes in industry and academia. The main weakness of this small-scale review is that the search for publications was limited to only one source (the IEEE Xplore[6] database) which may not be representative to the research field under study. For this reason, the conclusions drawn from the study were considered indicative rather than conclusive.

[6] http://ieeexplore.ieee.org

II

PART II—CONTEXT

In theory, there is no difference between theory and practice.
In practice there is.
— Yogi Berra

5 CHARACTERIZING MODEL USAGE IN EMBEDDED SOFTWARE ENGINEERING: A CASE STUDY

This chapter presents a case study in which the use of models in automotive software development was characterized. The study was conducted in order to gain a deeper understanding of the importance and cost of applying various modelling notations. The chapter was previously published as

> Mellegård, N., Staron, M.
> *"Characterizing Model Usage in Embedded Software Engineering: A Case Study"*
> Published at 8th Nordic Workshop on Model Driven Software Engineering (NW-MoDE), Copenhagen, Denmark 2010

5.1 INTRODUCTION

Model-driven engineering (MDE) (Atkinson and Kuhne, 2003; Brown, 2004a, 2004b) has the goal of increasing development productivity and quality of software-based products by raising the level of abstraction at which the development takes place. Several studies have provided evidence showing that the application of MDE in large industrial projects has indeed improved productivity and quality of the products (e.g. (Baker et al., 2005; Heijstek and Chaudron, 2009; Staron, 2008; Weigert et al., 2007)), while

other studies have shown mixed results or indicate a lack of objective empirical evidence (Mohagheghi and Dehlen, 2008).

Industrial software development, however, rarely provides the possibility to go from code-centric to model-driven software development in a single step and to the full extent. One example of software development domains where this is not possible is automotive software development. The domain is characterized by the existence of a significant amount of legacy software and high interdependence between car manufacturers and suppliers of car components. This high interdependence requires precise specifications that have to provide possibilities for interoperating of software development practices at the manufacturer's side and the supplier's side. This, in consequence, means that a number of practices for using models are used in parallel—e.g. UML and Simulink. Since modelling notations differ in the degree of formality and quite often can be used interchangeably, it is important to optimize the cost of using these notations. The cost has to be balanced with the benefits that these models bring and perhaps not used in those parts of software development where code-centric approach is already very efficient.

In this chapter we describe the development process and explore the costs and efforts of using different modelling notations and different abstraction levels for specifying requirements and designing the software in the automotive domain. The case study was conducted at Volvo Car Corporation, within the department responsible for electronic and software systems in Volvo automobiles. The research question in the case study was:

> *What is the relative importance of models-related artefacts in a model-based software development process?*
>
> *What is the relative cost (in terms of effort) for developing models-related artefacts in a model-based software development process?*

These two questions are important from a pragmatic perspective—*when introducing model-based development, are we focusing on the right improvements?*

Addressing this research question provided the possibility to assess the costs of using different kinds of modelling notations in software development. The data was collected via document analysis and a sequence of interviews with architects and project managers. The main perspective of the results is the project managers'. The results show that there are a few

artefacts which require significantly more effort than other artefacts and that the perception of what is important is different from where the effort is spent.

The rest of the chapter is structured as follows: Section 5.2 summarizes the method used, Section 5.3 outlines related work, Section 5.4 presents the results of the case study and Section 5.5 discusses the results. Section 5.6 concludes the chapter.

5.2 METHOD

The goal of this study was to characterize the process of developing automotive software systems, with use of models as the storage of design information. The research approach consisted of two separate stages: we first conducted a series of interviews and document analyses in order to build a map of the artefacts created in the actual process—a *development domain model* (Mellegård and Staron, 2010a). We then used the domain model in the second stage intended to assess which artefacts were considered most important and which received the most effort.

The first stage (Mellegård and Staron, 2010a) was conducted at VCC (Volvo Car Corporation) over the course of 18 months and consisted of semi-structured interviews with function, system and component designers and document inspections. The stage specifically aimed at addressing the following research questions:

M1.1 Which models are used?

M1.2 Which modelling notations are used?

M1.3 What information is conveyed by the models?

M1.4 How much automation is used in the development of models?

M1.5 Which documents are created from the information contained in models?

The second stage (Mellegård and Staron, 2010b) consisted of interviews and followed the case-study design proposed by Yin (Yin, 2002). The subjects in that stage consisted of 6 function designers and function managers, in which we asked the subjects to distribute $100 among the artefacts corresponding to the amount of effort that was put into each. The main unit of analysis was the relative importance and amount of effort invested in the development artefacts, specifically requirements, different types of models and documents. We first presented and described the domain model showing the key artefacts we had identified in the development phase.

The purpose with this presentation was to provide the respondent with the context of the study and also to provide the opportunity to ask about and comment on the content and structure of the domain model—e.g. add concepts or change relationships among the shown concepts. We conducted this part as a focused interview (Yin, 2002). In order to ensure construct validity, we noted the comments and incorporated them in the following parts of the interview (in accordance with Yin (Yin, 2002)).

5.3 RELATED WORK

This paper contributes to our previous research published in (Mellegård and Staron, 2010a) and (Mellegård and Staron, 2010b). In (Mellegård and Staron, 2010a) we reported on a domain model showing the main artefacts developed throughout the entire development process, whereas we in this paper focus on one of the four identified phases—the function development phase. Furthermore, in (Mellegård and Staron, 2010b) the focus was on describing the case-study in which effort and importance of the artefacts in the function development phase were investigated. In (Mellegård and Staron, 2010b) the preliminary results of 3 interviews were included, whereas this paper includes the result from 6 interviews.

Mohagheghi and Dehlen (Mohagheghi and Dehlen, 2008) conducted a review of documented modelling experiences and concluded that there are few reported results of how the MDE scales to large system development. Furthermore, the paper concluded that there was often a lack of company baselines which resulted in subjective evaluations. Our case study intended to contribute to such a company baseline intended to be used to compare the development process with other companies.

In (Staron, 2008) we contrasted how effort was distributed between the different development phases in model-driven and code-centric projects within a company in the telecommunication domain. The results showed that the overall efficiency of the development process improved by adopting an MDE approach. The evidence for the analysis and design phase, however, was not conclusive. In this case study we examined the analysis and design phase, specifically with respect to the development artefacts created. Furthermore, in (Staron, 2008) we found that the implementation phase had the most improvement potential; however, in the automotive domain the majority of implementation is done by third-party suppliers. This raises the question of how an MDE approach can be used to improve the efficiency of the analysis and design phase. In this case study we have taken the initial step

by examining the distribution of effort among and the perceived importance of the artefact developed in the phase.

Bollain and Garbajosa proposed in (2009) an extension to the ISO/IEC 24744 (ISO/IEC, 2007) meta-methodology standard. Their purpose was to extend the standard in order to better fit a document-centric (referred to as *code-centric* in this paper) development process. Our research has similar goals, but instead focuses on the use of models within an officially non-MDE development process. The case study reported in this paper provides evidence of which artefacts were considered the central ones, which in turn indicated where the focus of a modelling meta-model may be.

Broy (2006) outlined the challenges in automotive software engineering, and specifically outlined a structural view on the development process which he called a comprehensive architecture. Our case study focused on one particular part of this architecture, namely what Broy refer to as the *Design level* (which we refer to as the *function development phase*). Moreover, Broy concluded that although models were used throughout the automotive software development process, their use was fragmented. This meant that the benefit of having a coherent model chain—such as automatic artefact generation and traceability—was lost. Our case study contributes with empirical evidence that characterizes the development process—with regard to effort and importance of the constituent artefacts—which we hope will contribute with further evidence of how such an integrated modelling chain can be created in an optimal way.

Heijstek and Chaudron (2009) reported on an empirical study regarding model size, complexity and effort in a large model driven process, but whereas their study included the investigation of effort distribution among categories of development activities for the whole development process, our study focused on how effort was distributed among the artefacts produced in one of the development activities, and specifically on the effort distribution among types of models and requirements. Moreover, Heijstek and Chaudron reported on the importance and centrality of models in a pure MDE project, whereas our case study is conducted at a company which does not use an MDE approach. We elaborate on this in section 5.5.

Niggemann and Stroop (Niggemann and Stroop, 2008) described different kinds of models commonly used in the development of automotive software and their purposes. In particular, they identified two fundamentally different types of models used for the purpose of function development (algorithmic solution) and system development respectively. In our case study, our intention was to contribute with empirical evidence of the perceived

importance and the estimated amount of effort spent on the models and other artefacts in the development of automotive software.

5.4 RESULTS

5.4.1 Development Domain Model

The prescribed development process at VCC was a document-centric milestone driven process complying with the standard V-model. The process consisted of three main phases of development: the strategic phase, function development phase and the implementation phase in which the system and the physical components were specified. The prescribed process, however, stipulated what shall be delivered at each step of the development process, but not precisely how those deliverables shall be developed. In our domain model, shown in Figure 7, the top half represents the prescribed process and the bottom half shows the artefacts we found to be developed in the de-facto process. The domain model is described in the following sub-sections.

The Strategic Phase

The activities in the strategic phase (top half of Figure 7 as *Product Planning*), were focused around product planning and market positioning. The intended result of the phase was a feature list describing the planned vehicle model, including its feasibility in terms of required resources. Based on the outcome of this phase the decision was made whether the vehicle kind (e.g. a Volvo V50, V70) will continue into development or not.

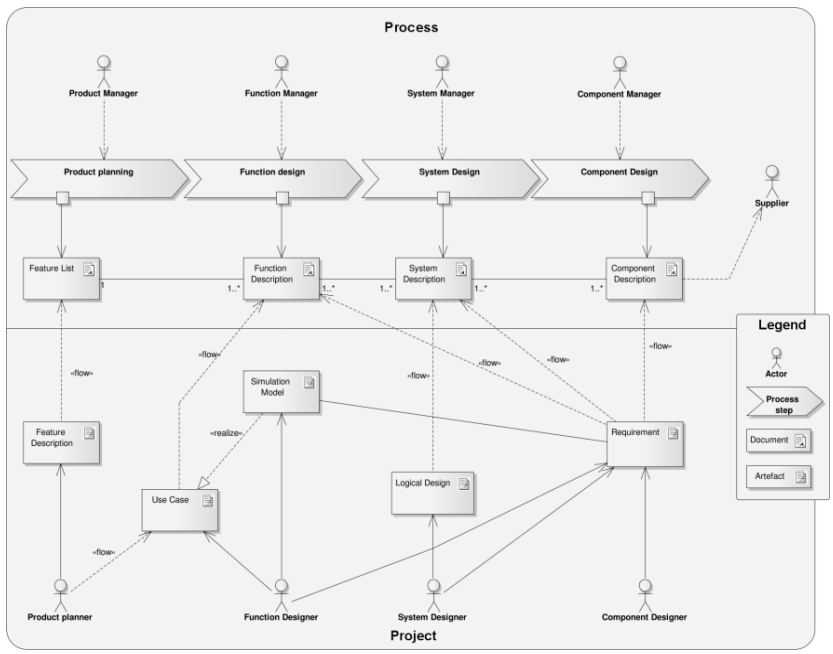

Figure 7. Project structure; process stages and their main deliverables. Please note the simplicity of dependencies in the prescribed process (upper half) and the complexity of dependencies within the projects (lower half).

In this phase the main stakeholders were product managers who work at the vehicle level. They consider such aspects as available technical solutions, competitor's products, available resources in terms of time and cost. Initially, only high-level properties were of interest to the product managers, for example "*the model shall be among the top 5 in the premium vehicle segment with respect to infotainment systems*".

During this phase the high-level properties were analysed and broken down into the features the vehicle model should have in order to fulfil the properties.

The example above shows that the descriptions here is high-level and that there was a need for a lot of intermediate steps in order to identify new functionality and which parts of that functionality would be realized in the software part of the car.

The Function Development Phase

The activities in the function development phase were focused on designing and specifying the behaviour of each separate function that together make up the feature, e.g. the climate control feature contained functions such as front seat heating and air ventilation or air condition (depending on the configuration of the vehicle model). The challenges addressed in the phase revolved around developing control algorithms that satisfy the desired behaviour.

The main input to the function development phase was a feature description (shown in Figure 7 as *Feature List*). Feature was a high-level description written from the end-user perspective. It described such properties as what functionality shall be offered, performance requirements and technology if it was of importance to the end-user. Additional input could be documentation such as function and system descriptions from similar previous systems. Such additional input was not present in all projects, only those where the documentation from similar features or previous versions existed.

During the function development phase, the high-level Feature was broken down into a number of separate Functions. Characteristics such as complexity of the feature, and dependencies on other features determined how the feature was decomposed into functions. Each function was specified in a Function Description document, shown in the top half of Figure 7.

The function description document began with one use-case that described the function from an end-user perspective. The use-case was intended to give an overview of the function, which formed a natural introduction to the document. However, this high level description was also important for validating the function against the feature description in the strategic phase.

The use-case was refined into detailed functional requirements, which was commonly done using models such as state machine diagrams with detailed requirements written for the states and the transition between states. These models and requirements are shown as the artefact *Requirement* in the bottom half of Figure 7 and as *Specification model* and *Requirement* in Figure 8.

The procedure of refining the initial use case into class diagrams, state machines and detailed requirements generally involved creating a simulation model (shown as the artefact *Simulink* in Figure 7). Simulation models were created for certain functions (mainly more complex ones) and based on the intended functional behaviour specified by the use case (shown in Figure 7 as the «realize»-relationship between *Use-Case* and *Simulink*). The purposes of the simulation model were found mainly (i) to assist the function developers in learning to understand the requirements, and (ii) to validate the behaviour

of the function with customer representatives at an early stage. The purposes for creating the simulation models were found to range from merely illustrating intended functionality to formal models from which production code could be generated. Consequently, many different tools were found to be used, from formal ones such as Simulink to informal ones such as MS PowerPoint and video sequences.

Close to the end of this phase the function designers together with system architects created an initial logical design for the function in form of a UML class diagram, in which functional responsibilities are assigned to logical components and their dependencies are shown. The logical design was further refined in the subsequent implementation phase. The function development phase is shown in more detail in Figure 8.

The Implementation Phase

The activities in the implementation phase were concerned with designing and specifying a system solution that satisfied the required behaviour of a set of functions (or parts of functions) on a particular physical platform—shown in the top half of Figure 7 as the many-to-many relationship between the documents *Function Description* and *System Description*.

The main input to the implementation phase according to the prescribed process was a set of function descriptions (shown in the top half of Figure 7 as *Function Description*). Furthermore, we found that simulation models (if available) and architecture specifications as well as documentation from previous version of the system provided additional input to the phase.

The implementation phase was divided in (i) system design and (ii) component design. The system design activity focused on designing and specifying a system that fulfilled the requirements from a number of functions on a specific vehicle platform, whereas the component design activity focused on deploying the system requirements on a set of physical nodes (as determined by the vehicle platform architecture). As shown in the top half of Figure 7, the realization of a function could be distributed among a number of systems, and a system could be deployed on a number of physical nodes. Normally, however, a feature (i.e. all the functions within the feature) was realized by one system, and that system is in turn deployed on a set of nodes. It was, however, common that a node contained functionality from more than one system, e.g. a sensor may provide data to multiple functions in multiple systems.

Given the above relationship between functions and systems we could see that the System Description document contains all requirements from functions that were realized by the system which is being described. The

system description document begins with listing all incoming requirements realized by the system—functional as well as non-functional. The document then presents the system from a logical, electrical, mechanical, and deployment view.

An important artefact in this phase was the Logical Design—a class diagram showing the logical view of the system, shown in the bottom half of Figure 7. The design could initially be identical to the logical design in the preceding function development phase. The system design, however, refined the logical design by adapting it according to the specific platform on which the function would be. The Logical Design was composed of a number of logical software components. These logical components were considered atomic units of functionality with a well-defined interface in the form of input and output data. These data would later be translated into signals sent on the communication busses on the physical platform. The model was used when mapping a logical view of the function to the platform on which the function will be realized. The system description document contained detailed requirements for each of the components, as well as for the communication between the components.

Logical software components were then mapped to Software Components (as defined by AUTOSAR (AUTOSAR, 2011)), which were deployed on a physical node, available on the target vehicle platform.

All requirements affecting the software units deployed on the physical node were collected in the Component Description document. The Component Description document was used as the specification when commissioning the node from a supplier (as shown to the right in the top half of Figure 7). Furthermore, simulation models could also be included in the Component Description document. In many cases, however, only parts of the models were shared with the suppliers due to confidentiality reasons. As the supplier was only provided with a partial view of the function, it added to the need for the requirements documents to be complete and correct.

Summary of results

M1.1 – What models are used?

We found that two types of models were mainly used (i) models-as-specification (which could be categorised as prescriptive models according to (Ludewig, 2003)) and (ii) models-as-implementation (which could be categorized as descriptive models according to (Ludewig, 2003)).

Specification models—such as use-cases, logical designs, and state machines—were used mainly to provide structure to requirements and to

serve as a communication medium for requirements, functional as well as non-functional.

Simulation models were used to create functional solutions that fulfilled the behavioural specification of a function. We found functions in which the simulation models were used throughout the whole development process. We also found evidence of code generated from the models which served as the implementation integrated into the supplier provided nodes. However, we also found systems in which simulation models were only used within the development process to explore possible solutions, and as a support for the engineers to learn to understand the requirements of the function.

M1.2 – What modelling notations are used?

Specification models were represented using a subset of UML: use-cases, classes and state machines. Formal simulation models (models-as-implementation) were mainly created with modelling tools with proprietary notation, such as Simulink, Statemate and Stateflow. We have however, also found cases where non-formal notations have been used (e.g. Microsoft PowerPoint and video sequences).

M1.3 – What information is conveyed by the models?

Specification models at the function definition level (e.g. use-cases and state machines) prescribed the required behaviour of the function. The specification models in the implementation prescribed mainly how responsibilities were partitioned between logical design elements, and then deployed on physical nodes. The simulation models used in both the function development and implementation phases, described a proposed solution that corresponds to the prescribed behaviour. We found simulation models that were only used internally in order to elaborate the requirements and specification models, but also found simulation models that were used as basis for generating production code.

M1.4 – How much automation is used in development of models?

We found two main cases where automation was used (i) to generate the required bandwidth on the platform communication busses from the deployment view based on UML class diagrams, and (ii) to generate production code from simulation models. The latter, however, was only done for some functions.

M1.5 – Which documents are created from the information contained in models?

We found that use-cases were included in all document except the component description. Logical Design was an important part of the System Description, but initial versions of the design have also been found in the Function Description document.

Although central to the development, simulation models were usually found not to be included in the official documentation. Rather, they existed as separate artefacts alongside the function documentation. Furthermore, in the cases where simulation models were provided to the supplier, it was found to be common to only provide the supplier with the parts of the model—due to confidentiality reasons.

5.4.2 Effort Distribution and Importance

In the case-study described in the previous section we found that there was differences between the prescribed process and the way products are actually developed. Whereas the prescribed process is document-centric and organized around milestones, the actual development process makes extensive use of models that are evolving throughout the process as advocated by MDE. This raises the question whether the distribution of effort and importance of models and other artefacts in our case is different from an MDE project.

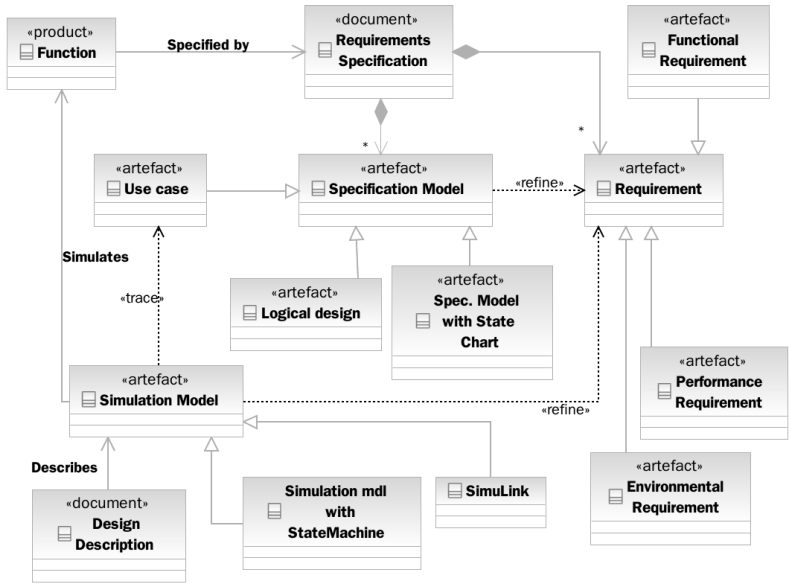

Figure 8. Artefacts, documents, and software products in the function development phase

In this case study, we collected evidence intended to address the following research questions:

M2.1 *What is the relative importance of the model-related artefacts in the software development process?*

M2.2 *What is the relative cost (in terms of effort) for developing model-related artefacts in the software development process?*

In this case study we focused on the activities within the function development phase (as described in *The Function Development Phase* on page 40), as the main software development takes place in that phase. In the interviews, we used the diagram in Figure 8 to depict the main artefacts and their relationships.

Proposition 1 – Importance of Artefacts

We had initially anticipated that models would be the most important artefacts, as they are generally more expressive than text. However, we found that it was rather the level of detail and the need to express requirements

unambiguously that was considered most important from the project manager's perspective. We found that the models were—from the project manager's perspective—mainly seen as support for the requirements; specification models were used to provide structure to the requirements or to provide them with a context. In particular, the simulation models were used to explore possible algorithmic solutions in order to learn to understand what the detailed requirements were.

Three of the subjects stated that use-cases were among the most important artefacts with the justification that they provided a high-level and easy to understand description of the function—i.e. as high-level requirements. However:

- One of the subjects stressed the fact that it is the contents that is important, not the modelling notation itself
- One subject explicitly stated that the use-case notation alone cannot be used as requirements provided to the third-party suppliers, as they leave too much room for interpretation

These observations were also supported by statements from most subjects, that detailed requirements regarding the environment of the function were the most important information—for example requirements on interfaces with sensors, actuators and other auxiliary components that the function depended on. The justification provided by the subjects was that "*without having these requirements for interfaces clearly and correctly stated, one would risk developing the function based on the wrong assumptions*".

Interestingly, the simulation models—which can be considered as the implementation of the function and hence the main product to be developed in the phase—were not considered to be among the most important artefacts. One subject stated that "*the most important thing is to agree on requirements regarding the environment of the function; the simulation model itself can easily be reworked, thus it is not that important*". However, it was also stated that traceability of these interface requirements between the specification and the simulation models was important. For instance in order to identify how the simulation model would be affected by changes in the interface specifications, as well as to be able to evaluate whether the simulation model complied to the interface requirements.

Proposition 2 – Effort Distribution

From the normalized result of the $100 technique, shown in Table 2 and summarized in Figure 9, we found that more than twice the effort was invested in models as compared to requirements. This finding supported our

anticipated result that the artefacts considered most important were not the same ones the most effort was spent on. However, we had not anticipated that the difference would be as big as it was in reality. Furthermore, it demonstrated that although the overarching development process was document-centric, on a project level models were considered to be more central.

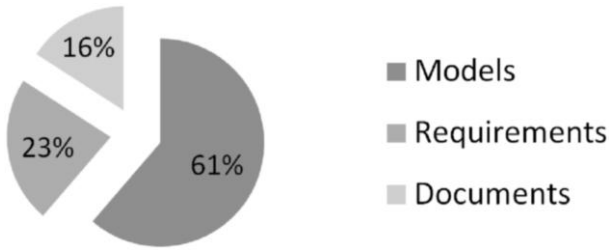

Figure 9. Distribution of effort among types of artefacts

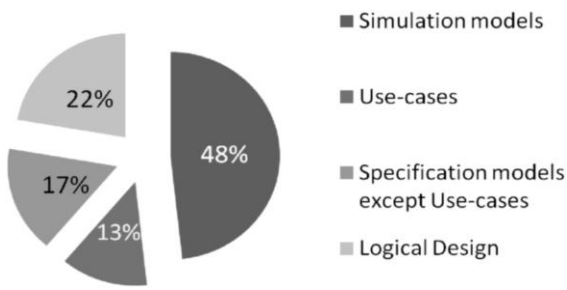

Figure 10. Distribution of effort among types of models

In particular, we have found that the majority of effort was spent on one of the artefacts as shown in Figure 10—simulation models. The reason for that was found to be

(i) They were used to explore the requirements of the function and assist in making them more precise, and

(ii) The simulation models were commonly used to generate the production code—the models could be considered as being the implementation, i.e. being the product to be developed.

Table 2 also shows that only between 24% and 55% of the effort was spent on developing the simulation models, which means that there was 45% to 76% overhead in the process of developing the production code. This large difference was not expected.

Interestingly, all subjects believed that the distribution of effort they had given would not be representative to other projects at the department; one subject stated "*you should probably take my numbers with a grain of salt, as I don't think my experiences are very similar to others*". However, as Table 2 shows, the distribution of effort between models and requirements, as well as between simulation and specification models was very similar.

When discussing the results with the interviewees and their managers after the study, their reaction was that this was against their conceptual model—i.e. they expected more effort to be spent on the requirements than it is now. This called for a subsequent (planned for future) study about detailed optimization techniques how to improve the throughput of the process. Possible targets for optimization is the multitude of different modelling techniques and notations used that are compatible to a limited degree.

Similar observations and how such challenges were overcome when Motorola successfully introduced MDE in their process are reported in (Weigert et al., 2007).

5.5 DISCUSSION

Model Driven Engineering advocate a systematic use of models and automated model transformations in software development projects. In our case study we have found that the studied organization applies the principles of MDE, but their process is still document-driven. We have found that the use of models is fragmented and there is no controlled sequence of models and transformations in the processes at the organization. This finding is similar to Broy (Broy, 2006) who found that even if models play a central role in organizations, their use is sometimes fragmented and notations are used inconsistently. That situation seems to be similar to the situation in the aerospace industry several years ago (Plant and Tsoumpas, 1995) and was not uncommon in the software industry more recently (Grossman et al., 2005).

However, contrary to the early adopters of MDE which we studied in our previous work (Staron, 2006), the goals for adopting modelling at VCC are different than others. In the previous study the main reason to introduce models was as a tool for communication, whereas the use of automation was not prioritized. In the case study at VCC we found that automation was the main goal for adopting models and therefore the modelling notations and

tools used in the implementation phases have the ability to generate source code from models. This finding indicates that the goal of adopting MDE at VCC is the same as the goals of MDE itself—to raise the abstraction level and increase productivity.

Furthermore, the ability to create executable models were considered important as such models were commonly created during the function development phase as the functional specification—this fining was also contrary to our previous study in which models in the initial phases were used mainly for communication and documentation. We also found that UML models were used together with other modelling approaches. We have seen that it was common to use different approaches for different purposes—whether of legacy reasons, choice of most appropriate tool for the task or because widespread use in the supplier chain. This finding was in line with Farkas et.al (Farkas et al., 2009) in which they proposed to incorporate a variety of artefact types with one modelling approach.

With a document-driven process—the studied development process corresponds to the *code visualization* level according to Browns modelling spectrum (Brown, 2004b)—one would expect documents to be the most important artefacts, the most effort intensive or both. Figure 9, however, indicate that the documents themselves were not the most effort intensive, nor did any of the subjects consider the documents very important. Nevertheless, it seems that the code visualization level will be applied in the nearest future due to the fact that documents are widely accepted by the suppliers as specifications, whereas models are not.

Works such as (Bollain and Garbajosa, 2009; Das Beauftragte der Bundesregierung für Informationstechnik, 2009) aim at improving the efficiency of document-driven processes by focusing on how documents and their consistent parts are managed within the process. From our findings, however, documents do not seem to be considered very important or effort-intensive. This would suggest that efforts to improve the process efficiency by focusing on the management of the documents themselves may not yield as significant results as focusing on artefacts within the documents and how they relate to each other. Furthermore, the distribution of effort between models and other types artefacts in our study was similar to the distribution found in studies of pure model-driven projects (Heijstek and Chaudron, 2009), which suggests that pure model-driven projects and other types of projects where models are used extensively may share similar properties. In model-driven projects, however, the models are used to generate other artefacts and to establish automatic traceability between artefacts. In our study, we found limited use of model transformations and automatic

traceability. This indicates that even though there was a similar pattern in how effort was invested with respect to models and other artefacts, the non-model-driven project did not use their models to their full potential.

As can be seen in Figure 10, simulation models were considered the most effort-intensive type of artefact. This was expected, as this type of model would often eventually be used as the implementation; the code deployed on the nodes was to a large extent generated from the simulation models. However, between 45% and 76% (see Table 2) of the effort was spent on artefacts other than the simulation models, which further indicates that there was a large potential for cost savings.

Table 2. Effort distribution in percent per artefact as estimated by the project managers

	PM 1	*PM 2*	*PM 3*	*PM 4*	*PM 5*	*PM 6*
Function					9.1	
Models	(56.4)	(50.0)	(73.4)	(52.4)	(54.5)	(81.8)
Simulation models	39.7	30.0	26.7	23.8	36.4	54.6
Use-cases	7.9	20.0	20.0	4.8	0	3.6
State machines / other specification models except UCs	4.0		26.7			5.5
Logical Design	4.8			23.8	18.2	18.2
Requirements	(23.8)	(30.0)	(26.7)	(23.8)	(27.3)	(9.1)
Requirements (all types)	23.8	30.0	26.7	23.8	27.3	9.1
Documents	(19.8)	(20.0)		(23.8)	(9.1)	(9.1)
Design Description	11.9					9.1
Requirements Specification	7.9			23.8	9.1	

In (Niggemann and Stroop, 2008) Niggemann and Stroop note that the mapping between simulation and specification models (which they refer to as function and system models respectively) is very important. This was confirmed by the interviews, as most respondents stated that agreeing on a

modular design and the communication between modules early in the development cycle was important.

5.6 CONCLUSION

The objective of this study was to investigate how effort and the perceived importance were distributed among artefacts identified in the function development phase of software-based vehicle functions. In this paper we have reported on a case study in which we interviewed project managers responsible for a software based safety and security related functions.

The most interesting finding in this case study was that requirements were considered undoubtedly the most important artefact. However, it was on the behaviour models that the majority of effort was spent. It was also these models that were used to understand and refine the requirements. In particular, requirements concerning communication interfaces and properties of auxiliary components are considered very important. However, simulation models—which commonly were used by the third-party component suppliers to generate the production code—were not considered as important. In addition, we found that although the simulation models constitute the single artefact that receives the most effort, they only received between 24% and 55% of the total effort spent. Considering these models as the implementation—i.e. the actual product under development—this means that 45% to 76% of the effort is spent on overhead activities.

Moreover, we have in this initial study of a non-MDE project found that the amount of the total development effort invested in models is similar to the pure MDE project reported in (Heijstek and Chaudron, 2009)—61% (average) in our case and 59% in the MDE case. In the case of a pure MDE project, model transformation is used to a large extent, whereas we only found limited use in our study. This indicates that although the distribution of effort between models and other artefacts are similar to an MDE project, the models are not used to their full potential. This in turn suggests that there are improvement opportunities by examining how existing models can be used more efficiently.

5.7 ACKNOWLEDGEMENTS

This research is partially sponsored by The Swedish Governmental Agency for Innovative Systems (VINNOVA) under the Intelligent Vehicle Safety Systems (IVSS) programme.

III

PART III—LIDEC

It is easier to confess a defect than to claim a quality
— Max Beerbohm

6 A LIGHT-WEIGHT DEFECT CLASSIFICATION SCHEME FOR EMBEDDED AUTOMOTIVE SOFTWARE

This chapter provides an in-depth description of a classification scheme adapted from IEEE Std. 1044 for the development of automotive safety software. The chapter was previously published as:

> Mellegård, N., Staron, M., Törner, F.
> *"A Light-Weight Defect Classification Scheme for Embedded Automotive Software"*
> Technical Report 2012:04, ISSN: 1654-4870, Chalmers University of Technology, Göteborg

6.1 INTRODUCTION

Software reliability is of central importance in modern cars as software controlled systems are becoming increasingly pro-active—recent safety functions are, for instance, able to automatically apply brakes to avoid crashes or mitigate their effects. Car manufacturers (OEMs) need, in order to achieve reliability, effective ways to manage defects during development (in-process) and during run-time (e.g. fault tolerance mechanisms). For the in-process defects it is important to identify, analyse and remove defects which

could compromise the reliability of the cars. Furthermore, identifying patterns in the in-process defects enables effective detection and removal of defects, for instance by indicating which test activities to focus on. In order to identify such patterns, however, systematic and structured defect documentation is required.

Defect documentation and analysis is common practice in most software development organizations. Its benefits are further emphasised through the inclusion in process maturity models—such as CMMI (CMMI Product Team, 2010a) and SPICE (The SPICE User Group, 2011)—as they require systematic defect documentation, analysis and follow-up. Neither CMMI nor SPICE, however, specifies how such defect documentation and analysis is to be done. Companies thus have their own interpretations resulting in varying quality of defect documentation; for instance, ambiguous interpretation of data or subjective opinions of the reporter. Hence, there is a need for a structured approach to defect documentation.

There have been several approaches proposed on how to perform structured collection and analysis of defect information; e.g. defect taxonomies (Beizer, 1990), root cause analysis (RCA) (Leszak et al., 2000) as well as various defect classification schemes such as Orthogonal Defect Classification (ODC) (Chillarege et al., 1992), the HP scheme (Grady, 1992) and IEEE Std. 1044 (IEEE, 2009). Although shown to be useful these approaches were designed for specific contexts (Freimut et al., 2005) causing the need for adaptations (Wagner, 2008); such adaptations have been identified as one of the major challenges in applying a defect classification scheme (Freimut et al., 2005; Wagner, 2008). Specifically, defect classification approaches often assume full knowledge of the defects, i.e. have a source-code focus and assume ownership of the software components (Freimut et al., 2005). Consequently, such defect classifications schemes need adaptations to be applicable to organizations where software is developed by suppliers—a situation common in the automotive software domain: even though software components (e.g. ABS or collision warning system) are often developed by suppliers, the quality of the complete product—the car—is the responsibility of the OEM. The need to systematically analyse and follow-up on the quality of the supplied software components is, nevertheless, important.

Furthermore, defect documentation—however important—may be seen as a mainly administrative task that does not directly contribute to the end-product. Thus, the defect documentation approach taken should require a minimum of analysis effort in addition to what is needed to identify and remove the defect, while still providing the additional benefit of

characterizing the quality of the product and development process (Freimut, 2001).

In this paper we address the challenges of efficient defect classification by pursuing the following research question: "*How to efficiently support defect identification and resolution time by classifying in-process defects?"* The research question is addressed by investigating how a defect classification scheme can be adapted to the automotive software development context by studying the development of active safety features. The aims of our adapted classification scheme—the Light-weight Defect Classification scheme (LiDeC)—include:

(i) as existing classification schemes have a strong source-code focus, how can such a scheme be adapted to a development setting with limited insight into the source-code;

(ii) as adding additional workload on development teams may reduce the likelihood of adoption, how can a classification scheme be adapted to minimize its process foot-print in terms of required learning and classification time.

LiDeC was developed as part of a case-study at Volvo Car Corporation (VCC[7]) and initially evaluated with a sample of problem reports from a project finished a year prior to the study. As a result we present a defect classification scheme, compliant with the IEEE Std. 1044 (IEEE, 1996, 1993), specifically adapted for the development of automotive safety-critical software. Furthermore, an initial evaluation showed that developers quickly learnt to apply the classification scheme, and that the required time to classify a defect was substantially lower than with other approaches to defect documentation.

The rest of the chapter is structured as follows: section 6.2 provides the study with background, section 6.3 describes the method used, section 6.4 summarizes the results and the final sections conclude the chapter and outline future work.

6.2 BACKGROUND

This section provides the research presented in this paper with background: first a summary of the terminology used is outlined, then related work is presented, and finally aspects of developing software at the case company is presented.

[7] http://www.volvocars.com

6.2.1 Terminology

In this report the terminology defined in the IEEE Std. 1044-2009 (IEEE, 2009) is used. Specifically the following terms are used in the report:

- *Defect*– An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced (IEEE, 2009).
- *Failure*– (A) Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. (B) An event in which a system or system component does not perform a required function within specified limits.
- *Fault*– A manifestation of an error in software.

Figure 11 shows how these terms relate and what is within the scope of the IEEE Std. 1044.

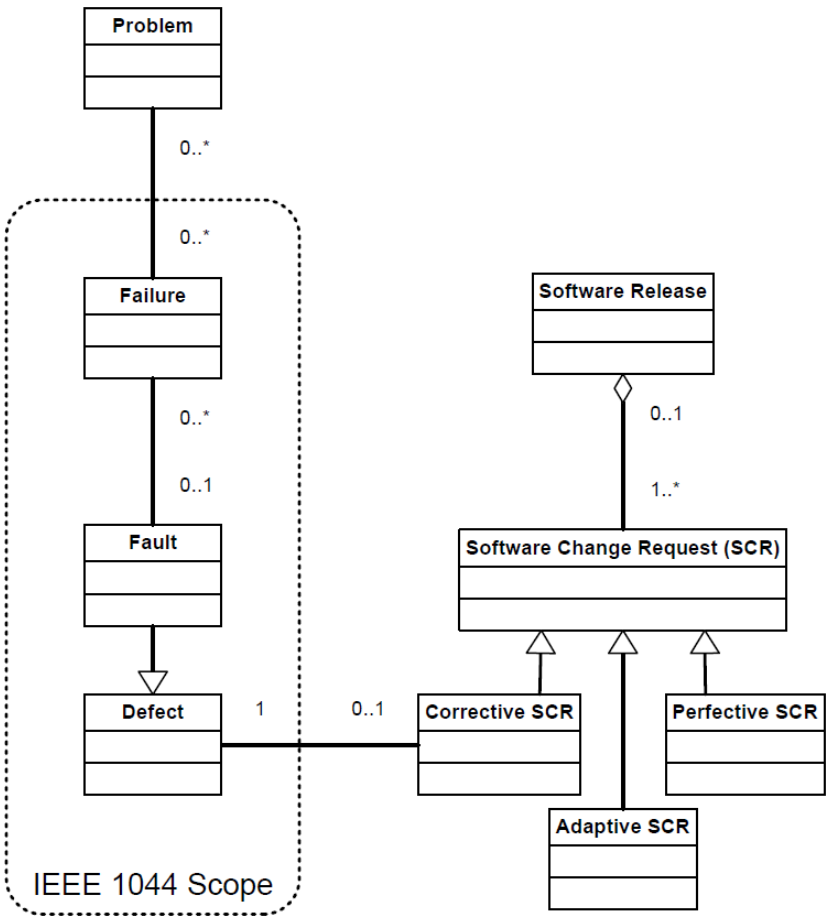

Figure 11. IEEE Std. 1044 concepts and their relationships (figure from IEEE Std. 1044-2009 (IEEE 2009))

As shown in Figure 11, problems are sufficient but not necessary conditions for the recognition that the software is failing to behave in a desirable manner. The failure may be caused by faults in the software, which in turn can be corrected by a software change request. These relationships are described in Table 3.

Table 3. Relationships between IEEE Std. 1044 Concepts (from IEEE Std. 1044-2009 (IEEE 2009))

Class/Entity pair	*Relationships*
Problem - Failure	A problem may be caused by one or more failures. A failure may cause one or more problems.
Failure - Fault	A failure may be caused by (and thus indicate the presence of) a fault. A fault may cause one or more failures.
Fault – Defect	A fault is a subtype of the supertype defect. Every fault is a defect, but not every defect is a fault. A defect is a fault if it is encountered during software execution (thus causing a failure). A defect is not a fault if it is detected by inspection or static analysis and removed prior to executing the software.
Defect – Change Request	A defect may be removed via completion of a corrective change request. A corrective change request is intended to remove a defect. (A change request may also be initiated to perform adaptive or perfective maintenance)

6.2.2 Related Work

This report is an extension of our previous work (Mellegård et al., 2012a). Whereas our previous work focused on the evaluation of LiDeC this report instead focus on the description of LiDeC; more specifically:

(i) The Background section has been extended with a terminology subsection
(ii) The Background section has been extended with more related work
(iii) The results has been extended with a new subsection with a comparison between LiDeC and IEEE Std. 1044
(iv) The description of the attributes in the results section has been extended with more details
(v) A full description of LiDeC has been added as Appendix A
(vi) A classification guide has been added as Appendix B
(vii) Two example classifications using LiDeC has been added as Appendix C
(viii) An IEEE Std. 1044 compliance matrix has been added as Appendix D

(ix) A mapping between attributes of IEEE Std. 1044 and LiDeC has been added as Appendix E

Defect reports are a valuable source of information about issues that arise in development: defect reports can reveal information about systematic problems with the development process such as the activities most prone to generating defects, or the efficiency of testing activities with respect to the number and type of defects they detect. Defect reports, however, are often used as a means to track and resolve the identified defect. But in order to systematically collect and analyse defect data there is a need to formalize the information collected about each defect. There are several proposed approaches which Wagner (Wagner, 2008) identifies as belonging to three main categories:

- *Defect taxonomies* which are categorizations of faults mainly related to the implementation. Wagner mentions examples categories such as wrong variable declarations and wrong variable scope;
- *Root cause analysis* (RCA) which is a more detailed approach. RCA not only analyses the fault itself, but also why the fault was introduced. The goal of RCA is to identify the root causes and eliminating them, thereby preventing similar faults from being introduced in future projects;
- *Defect classification* is an approach in which data is collected about the defect in a similar manner to both defect taxonomies and root cause analysis, but does so in a more coarse-grained manner.

Defect taxonomies are focused on the implementation and do not provide support for analysing what measures to take to prevent or mitigate any systematic issues it may reveal. RCA, in contrast, is focused on identifying why the identified defect was introduced into the system. RCA, however, is considered to be effort intensive and its cost/benefit is unclear (Wagner, 2008). Defect classification, on the other hand, aims at reducing the effort required to analyse a defect while still retaining the power of analysis—such as what types of defects are most common, which artefacts are most prone to defects. The approach taken is to gather a wider but more coarse-grained range of data. In this paper we have chosen to adapt a defect classification scheme given our goal of small process foot-print. Defect classification schemes are discussed in more detail in the following subsection.

In their paper Li et al. (Li et al., 2012) present experience from adapting existing issue tracking systems at two companies. The adaptations resembles our work as the pre-existing issue tracking system were mainly intended for in-process progress tracking of defect resolutions and resource management.

Their justification for adapting the issue tracking systems included inadequately designed attributes and attribute values which made the collected issue data poorly equipped for use as software quality assessment and software process improvement—data was entered inconsistently or omitted, resulting in the assembled data "behaving largely as an information graveyard" (Li et al., 2012). By redesigning the issue tracking systems—incorporating parts of ODC (Chillarege et al., 1992) and the IEEE Std. 1044—Li et al. were able collect higher quality data and use that data to point out improvement targets in both companies studied. Furthermore, follow-up analyses conducted after the process changes were able to detect improvement in terms of lower number of defects.

The work presented in this paper complements the scheme presented by Li et al. in that our classification scheme targets a development context in which code to a large extent is written by sub-contractors and where the sub-contractors own the source-code; thus, limiting the possibilities to analyse the exact nature of the defects. Furthermore contrary to the work presented by Li et al., where the classification scheme had to comply with legacy issue tracking systems—attributes were added to or modified in already existing defect databases—we had the opportunity to work alongside the team setting up a new issue management system. As a result, LiDeC is compliant with the IEEE Std. 1044. In addition, Li et al. identifies a number of lessons learnt that are of great interest in our work, as we currently are in the process of incorporating LiDeC in a new issue tracking system at our industrial partner.

Dubey (Dubey, 2012) reports on a case-study applying ODC scheme to a project developing embedded systems in which a substantial amount of software was developed by suppliers. They concluded that the classification scheme's focus on source-code required it to be adapted. Specifically, the attribute defect type needed adaptation. In their case there were many defects classified as "*Functional*" defects leading them to propose additional attribute values related to the design phase (as ODC originally only contained one value: "*Functional defect*").

In (Leszak et al., 2002, 2000) Leszak et al. report that conducting RCA on up to a year old defect reports in a distributed, component-based development process required on average 19 minutes per defect. Leszak et al. also concluded that analyses conducted in-process when detailed knowledge about the defects can easily be recalled would further reduce the required effort.

Cavalcanti et al. (Cavalcanti et al., 2011) investigated the problem of identifying duplicate defect reports in a number of private and open source projects. Specifically, they examined the amount of time required to analyse a defect to determine whether it was a duplicate or not. The time required

varied in the projects they examined from 5-10 minutes per defect to 20-30 minutes, with an average of 12.5 minutes. Furthermore, based on the size of staff and amount of defects reported, Cavalcanti et al. calculated that on average 48 man-hours per day was spent in search of duplicate defects in the examined projects.

Software reliability growth models (SRGM) (Kan, 1995) estimate software reliability by statistically correlating the cumulative number of defects discovered to a known function (Wood, 1996). SRGMs can be used to predict the number of residual defects in a product. SRGMs, however, do not provide any data about the type of defects, or in which part of the product they are likely to occur. Consequently, SRGM provides limited guidance as to which testing activities should be focused on to detect the yet unknown defects. Defect classifications, on the other hand, provide more detailed information—e.g. about detection and injection phase, and type of defect—and can be more precise than traditional SRGM (Ploski et al., 2007).

There have been many methods on fault prediction proposed. Liparas et al. (Liparas et al., 2011) examine a statistical method for analysing what factors in a multivariate data set that are best suited for predicting the number of defects contained in a software module. In their paper, Liparas et al. use a set of complexity metrics—such as McCabe's cyclomatic complexity, lines of code and branch count—to predict the fault-proneness of the modules. The method described predicted whether a module is within a normal cluster or not—where a module not in the normal cluster would contain more than the standard amount of defects. Such information is valuable when assigning resources; more resources can be assigned to the modules that are most likely to contain defects. While the number of defects may be used to indicate the modules in most need to testing, it does not, however, provide the testers with any indications of what to test for, nor which modules contain the most sever defects. In fact, in the comprehensive systematic literature review where Hall et al. (Hall et al., 2011) examined 36 studies (from a selection of 2,073) on fault prediction models published between January 2000 and December 2010, they found that few studies differentiate between the faults predicted; for instance, only one (Shatnawi and Li, 2008) of the 36 studies used fault severity in their prediction model. In order to differentiate between defects, more nuanced data about the defects are needed; our work aims at contributing to a model for assembling such defect data, thus enabling prediction of additional defect attributes.

Defect classification schemes

Defect classification schemes define a set of attributes, where each attribute captures a specific aspect of the defect—e.g. how the defect was detected, its severity and type. Moreover, for each attribute the schemes typically provide a set of values that can be chosen from; this contributes to the efficiency as well as to the reliability of the classification. The most commonly referred (Freimut, 2001) classification schemes in literature are ODC from IBM (Chillarege et al., 1992), the HP approach (Grady, 1992) named *Defect Origins, Types and Modes* (Wagner, 2008) (here referred to as the HP scheme) and the IEEE Std. 1044 (IEEE, 2009).

Regardless of which classification scheme is applied, the main challenge is to select the attributes and attribute values which are relevant to the specific development context (Freimut et al., 2005). In (Freimut, 2001), Freimut provides a framework for developing and using classification schemes; this framework has been followed in the work presented in this paper. Furthermore, in (Freimut, 2001) Freimut provides a comparison between the ODC, HP and IEEE Std. 1044 classification schemes and a mapping between the attributes of the different classification schemes.

The HP Scheme

The approach taken by the HP scheme is to define only three attributes: Origin, Type and Mode. The *Type* attribute is dependent on the value chosen for the Origin attribute. This first requires analysis of when the defect was injected into the system before its type can be established. Furthermore, the HP scheme does not explicitly capture data about how a defect was detected (its trigger (Chillarege and Ram Prasad, 2002; Chillarege et al., 1992)); there is thus no attribute available to identify which testing activities are effective in detecting particular defect types. Moreover, applying the HP scheme makes it difficult to identify effective testing techniques and investigate how late and severe defects can be identified earlier as the scheme does not include attributes such as:

- Severity of the defect from an end-user perspective
- The method by which the defect was detected
- Timing of defect detection
- The cause of the defect

Such issues considered important for VCC, thus a wider range of attributes than provided by the HP scheme needed to be collected.

IEEE Std. 1044

The IEEE Std. 1044 and ODC, in contrast, define a set of failure and defect life-cycle phases each containing a number of attributes (independent of each other) that are to be recorded; the life-cycle defined by IEEE Std. 1044 is shown in Table 4. The phases represent the states the defect can be in: initially a failure is recognized, then investigated, which might lead to discovering the cause of the failure (fault), an action to resolve the defect is then planned and the possible impacts of the chosen action/resolution are analysed, and finally what was actually done to close the defect. The attributes that are to be recorded in each of the phases represent information about the defect that is relevant for that particular phase; the information would be required to understand/resolve defects regardless of whether a defect classification is applied or not. This matched our requirements for LiDeC in that we—in order to minimize the process foot-print—intended to capture what was currently tacit knowledge in the defect analysis process.

Table 4. IEEE Std. 1044 attributes (adapted from (Freimut 2001))

Life-cycle Phase	*Attribute Name*	*Attribute Meaning*
Recognition	Project activity	What were you doing when the defect occurred?
	Project Phase	In which life-cycle phase is the product?
	Suspected Cause	What do you think might be the cause?
	Repeatability	Could you make the defect appear more than once?
	Symptom	How did the defect manifest itself?
	Product Status	What is the usability of the product with no changes?
Investigation	Actual Cause	What caused the anomaly to occur?
	Source	Where (part of the system and its documentation) was the origin of the defect?
	Type	What type of defect/enhancement at the code level?
Action	Resolution	What to do to prevent the defect from happening again?
	Corrective action	What action to take to resolve the defect?
Impact Identification	Severity	How bad was the defect in more objective engineering terms?
	Priority	Rank the importance of resolving the defect?
	Customer value	How important is a fix to the customer?
	Mission safety	How bad was the defect wrt. project objectives or human well-being?
	Project schedule	Relative effect on the project schedule to fix?
	Project cost	Relative effect on the project budget to fix?
	Project risk	Risk associated with implementing a fix?
	Project Quality/Reliability	Impact to the product quality or reliability to make a fix?
	Societal	Impact of society of implementing the fix
Disposition	Disposition	What actually happened to close the anomaly?

6.2.3 Study context – Automotive Software Development

The case-study presented in this paper was conducted at the department developing active safety features—such as collision warning, lane departure warning and driver alert control—at Volvo Car Corporation (VCC). In the following subsection we describe the development of software for such features at VCC.

Development process

The development process of active safety features at VCC (Mellegård and Staron, 2010a; Mellegård, 2010) can at a high level of abstraction be visualized by the V-model (Pfleeger, 2001). As shown by the left leg in Figure 12: product requirements are specified on vehicle-level and then refined through the development process into (sub-)system requirements and design. The system specifications are further refined into requirements and design of the individual hardware and software components that will realize them.

The bottom of Figure 12 shows the implementation of the components which is often done by suppliers; VCC commissions a component from a supplier based on requirement and design specifications. As VCC may have limited insight or control over the implementation phase—the in-house development activities are to an extent limited to design, specification and integration testing—applying defect classification schemes that have code focus consequently presents a challenge.

Furthermore, the test phases are shown by the right leg in Figure 12: components delivered by the suppliers are tested on unit level, subsystems are integrated and tested and finally the whole car is tested; it is in this phase that defects are reported.

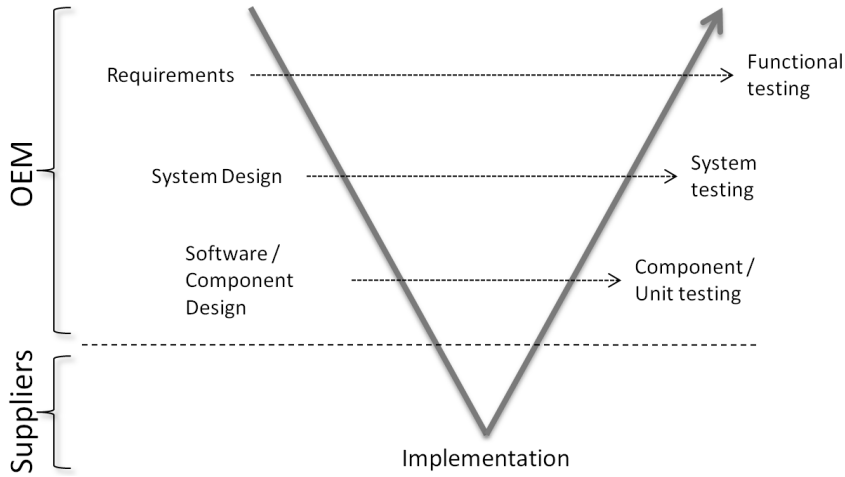

Figure 12. The V-model

Even though the overarching development process can be visualized by the V-model, in practice the process is better described as a federated development process (Bennet and Wennberg, 2005). As shown in Figure 13 development is iterative in three stages. In the first stage, corresponding to "*System Design*" in Figure 12, a system—e.g. collision detection or driver alert control—is designed and specified. The main focus of this stage is to develop algorithms that fulfil the high-level requirements. In addition to system requirements and design, the results from this stage may include executable models (e.g. Simulink models) that can be validated in simulated environments, e.g. using a test rig, or cars equipped with simulated hardware (e.g. dSPACE[8]).

[8] http://www.dspace.com

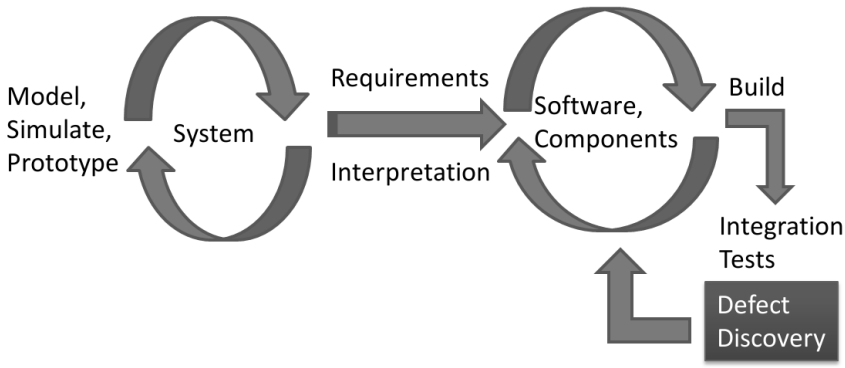

Figure 13.Federated development process (adapted from (Bennet and Wennberg 2005))

The second stage of development is shown in Figure 13 as "*Software, Component*" and corresponds to "*Software / Component Design*" in Figure 12. The focus of this stage is to decompose a system into individual software and hardware components that realize the system. The result of this stage is a set of component specifications that are used as base when commissioning components from suppliers.

In the final loop in Figure 13—corresponding to the right leg of Figure 12—software testing is done. As shown in Figure 13, the component design and test stages are intertwined. In practice, this means that the suppliers are involved from an early stage delivering a number of revisions of the components; each revision is tested by VCC, defects are discovered and corrected, and a new iteration is started. More specifically, the testing procedure done by VCC in each iteration of the third loop can be separated in three categories (illustrated in Figure 4 on page 12):

- *Component/Unit tests*. The algorithms—often in the form of executable models (Mellegård and Staron, 2010c)—are tested on unit level before being provided to the supplier. The supplier provides VCC with an implementation in the form of a component (e.g. optimized binary software component, or a hardware component with the software installed). VCC verifies that the component complies with the requirements;
- *System tests*. System tests are done on simulations of the system on a test rig using recorded data. The focus of this phase is on initial integration testing;

- *Functional tests*. Finally, function tests are done on builds of the system in a real car. Initial functional tests are run on test tracks, while in later project phases expeditions on roads are done. The focus of this test phase is on the whole vehicle, i.e. that high-level requirements are met (e.g. that features behave as intended from an end-user perspective)

In addition to the test activities done by VCC, suppliers conduct units test which may not be reported to the OEM.

In this case-study the focus was on defects discovered during the last two stages in Figure 13 (indicated in the figure by "*Defect Discovery*"). The reasons for this delimitation include the challenges associated with supplier implemented software—it is in this stage that the suppliers get involved in the development. Furthermore, initial analysis of the defect inflow—shown in Figure 14 as the defect backlog[9] (number of open defects over time)—revealed that there was a considerable spike in defects during the component development and integration testing phase (the start of which is shown in the figure as "*Software Phase*"). The increasing inflow of defects is expected as testing of supplier developed hardware and software—specifically integration testing—is conducted during this phase.

[9] The total number of defects has been scaled to 100 and the time scale has been removed due to confidentiality reasons. In addition, the time scale has been cropped (indicated by the ellipsis in the star and end of the curve) and does therefore not include the last phase leading up to start of production.

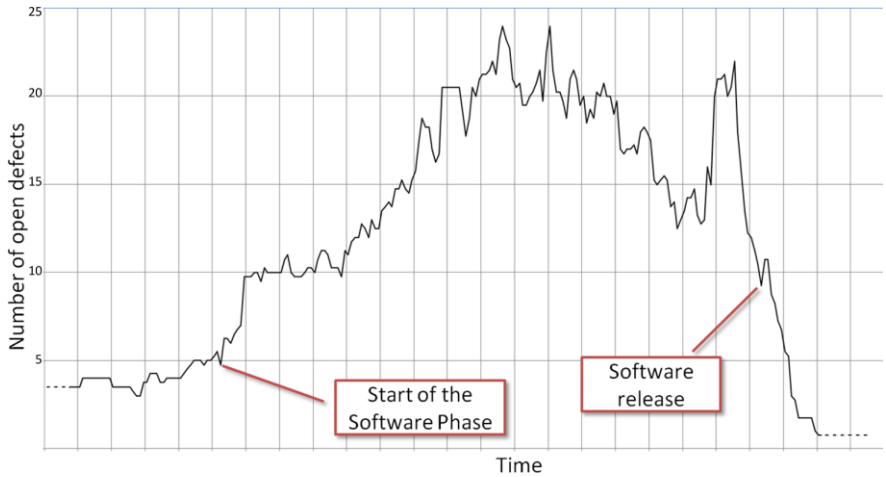

Figure 14. Defect backlog from the studied project

However, the increase in open defects (i.e. unresolved defects) during the software phase, and especially the peak close to software release (a major in-development milestone), raised the interest of our case company. Therefore, in the evaluation of LiDeC we analysed a sample of defects from the last peak shown in Figure 14.

6.3 METHOD

The research presented in this paper followed the case-study method described by Yin (Yin, 2002). The case-study method was considered appropriate as the applicability of the adapted classification scheme was of importance; adapting a classification scheme in the same context as it will be deployed would increase the chances of it being useful in that context. Specifically, we have used a single-case design by applying the classification scheme to defect reports from one project at our case company. The rationale of this was that the defect inflow profile from the project (shown in Figure 14) was considered representative by the developers—similar inflow profiles had been observed in other projects. Using the defect reports from the project, we developed an adapted defect classification scheme; in particular, we addressed the following main research question:

RQ *How to efficiently support defect identification and resolution time by classifying in-process defects?*

As the development context under study have specific properties (described in section 6.2.3), the main research question was broken down into:

L1 *In an automotive safety feature development context where source-code is often not available, how can a standard defect classification scheme be suitably adapted?*

L2 *As defect classification may often be considered an administrative task, how can the adaptation of a standard defect classification scheme be done to minimize required learning and classification time?*

As a practical guideline to adapting a defect classification scheme we followed (Freimut, 2001). The study was conducted in the following three stages:

Stage 1: Establish terminology. The aim of the first stage of the study was to establish a set of classification attributes using terminology aligned with the case company.

As a base for developing the classification scheme we used the IEEE Std. 1044 (IEEE, 2009) and its guide (IEEE, 1996). In this stage we began by choosing attributes and the set of values available for each attribute from the IEEE Std. 1044 that we—based on our previous research (Mellegård and Staron, 2010a, 2010b, 2010c)—found relevant for the specific development context at the company; for instance attributes related to customer value was not considered relevant for the development phase under study.

During two one hour-long interviews, we explained the initial classification attributes to the interviewee (project leader), and asked the interviewee to relate these attributes to the case company. We took notes during these interviews and refined the classification scheme according to these notes.

Stage 2: Tune feasibility. The second stage of the case-study aimed at streamlining the set of values each attribute could be assigned.

The stage consisted of a two hour long interview with a developer in which a number of defects were classified. The set of values available for each attribute was evaluated during the classification session, where for each attribute:

(i) If an attribute value was never used and the interviewee could not think of an example when the value would be used, the value was considered for removal;
(ii) If the interviewee did not consider any of the available values described the defect sufficiently, a new value was considered for addition.

As a result the final classification scheme was defined and depicted in the form of flowcharts with short questions providing a guide to arrive at the correct attribute value. Each attribute value was provided with a short illustrative example.

Stage 3: Evaluate scheme. The final stage of the study aimed at evaluating the efficiency and effectiveness of the classification scheme.

In this stage defect reports were classified according to the scheme. Four subjects involved in the project participated in six separate two hour long classification sessions (two subjects participated in two consecutive classification sessions). Three of the subjects were not involved in the previous two stages of the project.

The project used as case had finished one year prior to the study and contained over 100 problem reports[10]. All subjects involved in the study had been part of the project with the following roles: two developers, one tester and one project leader.

During the third stage of the study we were able to classify 22 defects. Of the 22 defects 12 were randomly selected from the last peak (as shown in Figure 14) and the remaining 10 from the rest of the project. This selection was done because of an expressed interest by members of the project to gain more insights in the defect peak. The results of the evaluation are reported in (Mellegård et al., 2012a).

6.3.1 Validity Evaluation

We have identified and grouped the threats to validity in our study according to recommendations of Yin (Yin, 2002):

- *Construct validity*– By basing our defect classification scheme on the IEEE Std. 1044 and by keeping careful notes on how to map concepts specific to our case to the standard, we consider that the threat to construct validity to have been minimized. Furthermore, as both the adaptation of the classification schemes and the evaluation was done using real defect data from an industrial project with the assistance of

[10] Exact number cannot be disclosed due to confidentiality reasons

the developers involved in the project, we consider the threat to construct validity to have been further reduced.

- *Internal validity*– As any interview study we anticipated some personal bias in the answers from the interviewees. In order to minimize this threat we triangulated the results by including multiple subjects in our interviews. In addition, a set of defects were classified by multiple subjects thereby allowing evaluation of the repeatability of the classification scheme; section 6.4.2 reports the results from this evaluation.
- *External validity*– There is a risk that the results are too specific to Volvo Car Corporation. However, as we documented and justified the modifications done to the IEEE Std. 1044 as well as described the particular development context of our case, we believe that our results can be generalized to similar contexts outside our specific case. Moreover, we consider the mapping between attributes of the classification schemes provided by Freimut (Freimut, 2001) to contribute to the generalizability of our classification scheme; e.g. the mapping between classification schemes enables analysis methods utilized with other schemes to be applicable to LiDeC as well, and therefore we believe that results are also comparable.
- *Reliability*– As part of the case study design, we have created a case study protocol which ensured that we conducted the study and collected the data in a consistent manner. By using this protocol, we believe that the study can be reliably reproduced.

6.4 RESULTS

The main challenge of adapting the IEEE Std. 1044 included tailoring the attributes relating to the fault and its resolution; specifically attributes in the phases *Investigation*, *Action* and, *Impact Identification* as the attributes in these phases have a strong focus on source-code aspects.

The results are reported below in two parts; first, the classification scheme is presented, and second, a comparison with the IEEE Std. 1044 is presented; for results from the initial industry evaluation of LiDeC, see (Mellegård et al., 2012a).

6.4.1 LiDeC

LiDeC captures – as shown by the scheme overview in Figure 15 – attributes from four phases of the defect life-cycle (IEEE, 1996, chap. 8.1) (described in more detail below as well as in Appendix A and Appendix B): the first phase captures information about the recognition of the defect, i.e. observing a deviation (failure) from intended or specified requirement (ISO, 2005); the second phase captures information about the underlying cause of the defect (referred to as *Investigation* in (IEEE, 1996)); in phase three information about the defect resolution is captured (referred to as *Action* in (IEEE, 1996)); and the last phase captures information about what was actually done about the defect (referred to as *Disposition* in (IEEE, 1996). Table 5 shows a comparison between the life-cycle phases of IEEE Std. 1044, ODC and LiDeC.

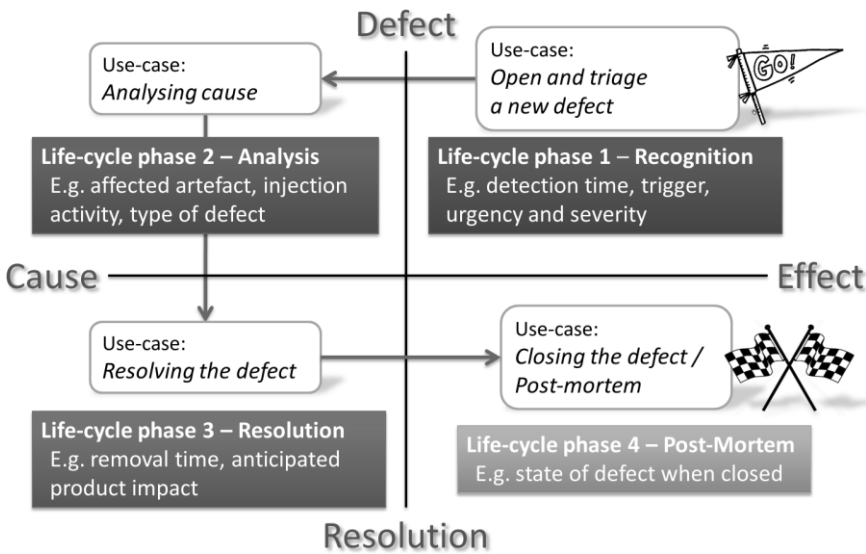

Figure 15. Overview of the LiDeC Scheme

Table 5 Mapping of life-cycle phases

Defect life-cycle phase	*IEEE 1044*	*ODC*	*LiDeC*
1	Recognition	Open	Recognition
2	Investigation		Analysis
3	Action	Close	Resolution
4	Impact Identification		
5	Disposition		Post-mortem

As can be seen by the number of attributes in each phase (described in Table 6 to Table 9 in the subsections below), the main focus of LiDeC is on recognition and analysis of a defect. The justification is that the later a defect is discovered the more costly its resolution tend to be (Boehm, 1981). In addition, as implementation is done mainly by suppliers, the most promising areas of process improvement lies in more efficient verification and validation. Consequently, the main focus of the classification scheme is on how defects are discovered, how the product is affected by them, and what types of defects they are. Analysing this information will contribute to understanding which phases of the development process contain the most improvement potential.

The phases of the classification scheme are aligned with the defect management process at the case company and directly correspond to the states a defect can be assigned: recognized, analysed, resolution proposed and post-mortem. The following sections describe the attributes of each phase. Appendix A provides an exhaustive list of attributes and their description and Appendix B provides a classification guide that was used in the case study. The classification guide in Appendix B contains a more detailed description of each attribute value along with typical examples expressed in the terminology of the company; the purpose is to maintaining consistency of the classification over time and between reporters. In addition, Appendix C contains classification examples.

Recognition

The attributes in the first phase of the defect life-cycle (shown in Table 6) relate to data about the discovery of a failure and its effects on the system in question, i.e. the manifestation of the defect. The attributes capture project related information:

- Timestamp of detection
- Resolution urgency
- End-user perceived severity of the defect, and
- How the defect affects the product, including whether ASIL requirements are affected (ISO/IEC 26262 (ISO/DIS, 2011)).

This information will be used in analysis, for example, to assess how timely the most serious defects are detected, or which activities are more effective in detecting defects.

Table 6 Summary of attributes in the Recognition phase

Attribute	*Question*	*Values*
Timing / Detection	When was the defect detected?	Date and project phase
Timing / Preferred	When should the defect have been detected (subjective)?	Project phase if different from Timing / Detection
Affects S/W	Does the defect affect software?	Yes / No
Detection Activity	What was done to detect the defect?	Inspection/Requirements, Inspection/Design, Unit test/In-house, Unit test/Supplier, System test/bench, Functional test/Test track, Functional test/Expedition Production/Manufacturing Production/Customer report
Urgency	How urgently does the defect need to be addressed?	Immediately, Next development release, Before start of production, Deferrable
Severity	How severe is the defect with respect to product quality?	None, Nuisance, Limited Functionality, Show-stopper
Effect	How does the defect primarily affect the product?	Capability/Undesired activation, Capability/Inactive on true positive, Capability/Other, Function Safety, Maintainability, Usability, Testability, Configurability
Functional Safety Impact	Does the defect have impact on a software component with ASIL-classified requirements?	Yes, No

Analysis

The attributes in the second phase of the defect life-cycle (shown in Table 7) aim at capturing data about the cause of the failure; e.g. in what work product and product component contained the defect, what type of defect it was, and which process step caused the problem.

Table 7 Summary of attributes in the Analysis phase

Attribute	*Question*	*Values*
Artefact	Which software work product contained the defect?	Req./Internal, Req./Cross-function, Req./External, Design model, Impl./Executable model, Impl./Code, Impl./Configuration params, Tool
Injection activity	When was the defect injected?	Specification, Design, Impl./In-house modelling, Impl./Suppl. mdl. transform., Impl./Supplier coding, Configuration
Component / Asset	Which design component contained the defect?	Internal (product) module name also identifying its version
Type	What type of defect was it?	Description, Data, Interface / Timing, Logic / Algorithm, Tooling, Tuning

Resolution

The attributes in the third phase of the defect life-cycle (shown in Table 8) aim at capturing data about the proposed resolution. As implementation specific details of the resolution may not be available, the attributes in this phase focus on capturing the cost of resolving the defect in terms of development effort. More specifically, to capture what impact a resolution would have on the product and on the process; the impact on the product is captured in terms of how much of the product would be affected by the

modification, and impact on the process in terms of amount of regression testing needed.

Table 8 Summary of attributes in the Resolution phase

Attribute	*Question*	*Values*
Removal time	When was the defect report closed?	Date and project phase
Product impact	What would the impact of a proper resolution be on the product?	None, Local (unit) modification, Multiple components, Funct. changes (re-design)
Required Verification Level	What level of regression testing would a proper resolution require?	None, Inspection, Unit test, System test, Expedition

Furthermore, the attributes of the *Resolution* phase capture data about a proper resolution of the defect. In practice, a defect could be resolved by means of workarounds (this data is captured in the final phase of LiDeC).

Post-mortem

In the last phase of the defect life-cycle the single attribute (shown in Table 9) records what was finally done to close the defect; to what extent the defect was resolved.

Table 9 Summary of attributes in the Post-mortem phase

Attribute	*Question*	*Values*
Resolution state	What was the final state of the defect when the problem report was closed?	Corrected (proper resolution applied), Workaround/Fix, Workaround/Product de-scoped, No Action/Deferred, No Action/Referred, No Action/Not found, No Action/No action

6.4.2 Comparison with IEEE Std. 1044

In the process of adapting IEEE Std. 1044, compliance with the standard was considered an important requirement (Appendix D shows the compliance matrix as proposed in (IEEE, 1996)). Retaining compliance with the standard contributes to the generalizability of the results—e.g. data collected with LiDeC and analyses conducted on that data should be comparable with IEEE Std. 1044 compliant data from other companies.

The main differences between the IEEE Std. 1044 and LiDeC are described below. The full mapping of attributes between the IEEE Std. 1044 and LiDeC is presented in Appendix E.

General Modifications

The main adaptation made to LiDeC consists of raising the abstraction level of the attributes, i.e. choosing attribute values that are less fine-grained than their IEEE Std. 1044 counterpart. Furthermore, the set of values available for each attribute expressed in the terminology of the company and provided with examples; for example the attribute "*Type*" has been shown to be problematic (e.g. "*to me everything is a logic problem*" (Freimut et al., 2005)); in LiDeC typical examples for each available value (see for instance Figure 42 in Appendix B) are provided.

Furthermore, attribute values that in IEEE Std. 1044 consisted of "Low, Medium and High" (e.g. *Project Risk* and *Priority*) has been replaced by more descriptive values (see, for instance, the LiDeC attributes *Urgency* and *Severity* in Appendix A and figures Figure 37 and Figure 38 Appendix B). This contributes further to making values less ambiguous by limiting the amount of interpretation needed by the reporter.

Moreover, supporting data items from the standard, e.g. cost and time estimations, defect reporter, developer assigned to the defect) has been omitted from LiDeC unless they have a specific purpose related to the analysis of the defect data (e.g. the LiDeC attribute *Component/Asset* is originally a supporting data item in the IEEE Std. 1044 *Action* phase). Such supporting data items, however, are assumed to be part of the company's normal issue tracking process as needed.

Added Attributes

The attributes described in the following subsections were added in LiDeC.

Timing/Preferred

The attribute was added to the *Recognition* phase in order to capture—at the time of detection—the reporter's subjective opinion of whether there was a

previous test phase in which this type of failure should have been uncovered. The intention of the attribute is to be able to gauge the fault-slip-through rates of the test activities.

Affects Software
The attribute was added to the *Recognition* phase in order to capture whether the defect has an impact on software. As the products developed are software intensive mechatronic systems, there may be defects that are not related to software; the purpose of the attribute is thus to be able to filter defects based on whether they affect the software. The definition of whether a defect affects software, however, is broad; see the example given in the classification guide in Figure 35 in Appendix B.

Component/Asset
The attribute was added to the *Analysis* phase in order to be able to evaluate the distribution of faults among components in the system. The attribute is originally part of the supporting data items of the IEEE Std. 1044 *Action* phase – thus whereas the IEEE Std. 1044 records the component(s) in need of modification, LiDeC records the component(s) containing the fault. This redefinition was made as the main focus of LiDeC is on capturing data about the defects rather than their solution; as the majority of implementation is done by suppliers, it is of more interest to the company to identify which components contain the defects rather than which components need modification (e.g. a workaround may require modifications to other components than the one containing the defect).

Removal Time
The attribute *Removal Time* was to the *Resolution* phase added in order to allow evaluation of defect longevity. The attribute is originally part of the supporting data items in the IEEE Std. 1044 *Action* phase.

Omitted Attributes
In this section the attributes available in the IEEE Std. 1044 that were omitted in LiDeC are listed. All omitted attributes are listed as optional in IEEE Std. 1044.

Suspected cause
While the *Suspected cause* attribute may provide valuable input during the process of analysis a failure, it was not considered important from LiDeC's point of view. LiDeC aims at capturing attributes of the defect itself, rather than speculations done as part of the defect analysis process.

Repeatability
The *Repeatability* attribute was omitted (as a separate attribute) in the LiDeC for the same reasons as the *Suspected cause* attribute (above). In LiDeC, the attribute is instead partly represented as an attribute value in the *Disposition* attribute of the *Post-mortem* phase; non-repeatable defects would be reported as *No action / Not found*.

Corrective action
The IEEE Std. 1044 attribute *Corrective action* records detailed data about what action was taken in order to resolve the issue. The attribute values proposed by the standard—such as revising the developing process or implementing a training program—would not generally be applicable to the context of VCC based on a single observed defect; rather such actions would instead be taken based on analysing defect data from a number of projects.

Furthermore, the exact action taken may not easily be identified as a substantial amount of defects relates to code, and code is generally produced by suppliers. Instead, LiDeC captures data about the estimated repercussions of the corrective action(s) by the attributes *Resolution Impact* and *Required Verification Level*.

Customer value, Mission/safety and Project risk
As the roles responsible for reporting defects may not have the necessary insight in, for instance, product planning the (explicit) attributes *Customer value* and *Mission/Safety* were omitted. Instead the data are in LiDeC implicit in the attribute *Severity*—i.e. how severely the defect affects the product from a customer perspective.

The attribute *Project risk* was omitted in LiDeC as a defect reporter may not have necessary knowledge of project planning (defects may be reported by suppliers, as well as various in-house testers) to be able to assess project risk. The risk can, however, be assessed by analysing the LiDeC attributes *Resolution impact* and *Required verification level* together with *Severity* and *Time of detection*; late sever defects that have a large impact on the product and/or require the more costly verification types would constitute a higher risk.

Project quality/reliability
The intention of the IEEE Std. 1044 attribute *Project quality/reliability* is to appraise the impact on the project quality if a defect is addressed. The attribute was omitted in LiDeC as the data was not considered relevant to study context—the subjects in the case-study had difficulty relating the attribute to the defects analysed, indicating that the roles responsible for

defect reporting may not have the necessary insight in project planning to be able to reliably assess the attribute.

The LiDeC field *Required Verification Level*, however, captures data with similar intentions: when addressing the defect how much additional re-verification effort would be required (also see *Project schedule* below).

Redefined Attributes

A number of IEEE Std. 1044 attributes have in LiDeC been redefined. The redefinitions have been done with care to retain the intention of the original attribute, but adapted to the context of VCC; e.g. by changing the available attribute values (for instance the *Symptom* attribute) or by using a different measurement unit than the IEEE attribute originally specified (for instance the *Project cost* attribute).

The following subsections describe the attributes that have been redefined.

Symptom

The IEEE Std. 1044 attribute *Symptom* has been redefined in LiDeC in that, whereas the original attribute captured more detailed data about the behaviour of the system, the LiDeC attribute *Effect* captures less fine-grained data about what quality aspect of the product—i.e. the product's "-ablities"—is affected.

Furthermore, the attribute values chosen for the *Effect* attribute has been tailored specifically for the context of active safety systems. For instance, the *Capability* attribute values (which would correspond to the values available in the *Symptom* attribute) has been subdivided into the two categories that are most relevant (the function triggering on a false positive and the function being inactive despite a true positive) as well as a catch-all third capability related value, see Figure 39 in Appendix B.

Product Status and Severity

Both the IEEE Std. 1044 attributes *Product status* and *Severity* are included in the LiDeC attribute *Severity*. Whereas the IEEE attribute *Severity* captures the severity of the fault, the LiDeC attribute *Severity* captures the severity of the failure (i.e. the manifestation of the fault)—as does the IEEE *Product Status* attribute. LiDeC's focus on the failure aspect of a defect rather than the fault causing it is due to the limited in-house implementation done, and the specific interest in the ability to evaluate test activities.

Furthermore, the attribute values available in the LiDeC attribute *Severity* have been defined to describe the impact of the failure in more objective terms than the original values proposed by the IEEE Std. 1044 (which are "Urgent, High, Medium, Low and None"), see Figure 38 in Appendix B.

Societal

The attribute *Societal* is in LiDeC covered by the attribute *Functional Safety Impact*. Whereas the IEEE Std. 1044 does not define the *Societal* attribute clearly, in the attribute *Functional Safety Impact* is specific with respect to defects that may cause harm. In particular, LiDeC attribute captures whether the defect have impact on a software component that has ASIL-classified requirements (as defined by ISO/IEC 26262 (ISO/DIS, 2011)).

Actual cause

Whereas the IEEE Std. 1044 captures data about the artefact that caused the defect, LiDeC captures data about in which project phase the fault was injected. This redefinition was made as it was, in the case-study, found that the attributes *Actual cause* and *Artefact* was treated identically. By referring to the activity causing the defect it was clearer to the subject how to use the attribute (see Example 2 Appendix C).

Type

The IEEE Std. 1044 attribute *Type* has a strong focus on source code, and provides a detailed set of attribute values. In the LiDeC case-study the attribute was found difficult to assign of two main reasons: i) as the access to source-code may be limited, identifying detailed data about type of defect is often not possible; ii) the detailed set of values proposed by the IEEE Std. 1044 were shown to make distinction between values difficult (this is also corroborated in (Freimut et al., 2005) by the quote "*to me everything is a logic problem*").

The approach taken in LiDeC is to substantially reduce the resolution in the available attribute values, and to provide each attribute value with a typical example in the terminology of the company (see Appendix A for a description of the LiDeC attribute and Figure 42 in Appendix B for examples of each attribute value).

Resolution and Priority

The IEEE Std. 1044 attribute *Resolution* captures data about both the urgency of the resolution and what type of resolution that will be applied (IEEE, 1996). In LiDeC, the urgency of resolving a defect is captured by the *Urgency* attribute while data about the type of resolution is not explicitly captured by LiDeC (instead, it is partly covered by the *Disposition* attribute).

Furthermore, whereas the IEEE Std. 1044 *Resolution* attribute relates to the urgency of applying a specific resolution, the LiDeC attribute *Urgency* relates to the urgency of removing a failure from the system; thus, the focus is on detecting and prioritizing the removal of the manifestation of defects

rather than on details of the actual resolution (as the resolution may be developed and applied by the supplier, the OEM may not have the necessary insight).

Project schedule
The IEEE Std. 1044 attribute *Project schedule* aims at capturing data about the direct impact of the resolution on the project schedule. In LiDeC this is instead captured in terms of amount of re-testing needed after applying a resolution; as verification activities constitutes a substantial amount of the development efforts, the attribute captured data with the same intention as the *Project schedule* attribute. Furthermore, the amount re-testing needed for a resolution is more straight-forward to assess for an engineer reporting the defect than objectively estimating the impact on project schedule.

Project cost
Whereas IEEE Std. 1044 attribute *Project cost* captures data about the cost of a resolution in terms of real money, LiDeC instead makes the estimate in terms of how much of the product will be affected by the resolution in the attribute *Product Impact*—the assumption is that the more of the product that is affected the more expensive the resolution will be. This redefinition was made as the engineer reporting a defect may not have sufficient insight into the budget or may have limited ability to make a reliable estimation of the cost of applying a resolution. Thus, LiDeC captures data with the same intention as the *Project cost* attribute but in more objective engineering terms.

6.5 CONCLUSION

In this report we have described the adaptation of IEEE Std. 1044, the IEEE standard for defect classification, to the automotive safety feature development context. The specific properties of the development context that influenced the adaptation include a strong reliance on supplier side implementation, which may limit the access and insight into the source-code. Furthermore, as defect classification does not directly contribute to the development of the end-product, it was considered important to adapt the classification scheme to minimize the time required to use the scheme while still providing the additional benefits of characterizing the defects. More specifically, we have addressed the research questions:

L1 *In an automotive safety feature development context where source-code is often not available, how can a standard defect classification scheme be suitably adapted?*

L2 *As defect classification may often be considered an administrative task, how can the adaptation of a standard defect classification scheme be done to minimize required learning and classification time?*

We addressed L1 by:

- Shifting the focus of the classification scheme from detailed aspects of the fault and its resolution to aspects of the discovery of the defect. As the implementation is mainly done by suppliers, most in-house process improvement potential lies in more efficient defect discovery activities. LiDeC reflects this by providing more detailed attributes in the *Recognition* phase (e.g. *Detection activity*, *Urgency*, *Severity* and *Effect*), while granularity of the attributes in subsequent phases have been reduced; e.g. the *Type* attribute is less granular than in IEEE Std. 1044 and detailed aspects of the resolution (captured by the IEEE attribute *Resolution*) has been omitted;
- Adapting attributes for safety specific purposes. In LiDeC the attribute *Functional Safety Impact* (which maps to the IEEE attribute *Societal*) records whether a defect impacts ASIL-classified requirements (ISO/DIS, 2011). In addition, the values of the Effect attribute (which maps to the IEEE *Symptom* attribute) was adapted specifically to provide a high-level characterization of safety feature problems (e.g. unintentional activation of a feature).

We addressed L2 by:

- Raising the level of abstraction of attributes. For instance, by providing only higher-level categories as values for the Type attribute
- Providing more descriptive attribute values, e.g. instead of *Low*, *Medium*, *High*, more descriptive values expressed in the terminology of the organization were used
- Providing attribute descriptions and values phrased using the terminology of the company
- Providing a classification guide with a flow-chart structure, and including typical examples for each attribute value
- Streamlined the attributes by removed or redefining attributes that required insights that the typical reported may not have (project schedule and risk). Attributes were redefined with care to retain the intentions of the original attribute but measured in more specific engineering terms; for instance, whereas the IEEE Std. 1044 attribute *Project schedule* aimed at appraising the direct impact on the project

plan, LiDeC instead appraises the estimated amount of re-verification (an activity that may have a large impact on the project schedule).

In conclusion, by putting focus on capturing data about the discovery of defects, streamlining the available attributes with respect to the expertise of the engineers reporting defects, and conforming to the terminology of the company, we have developed an IEEE Std. 1044 compliant classification scheme well suited to the development of automotive safety features at VCC. As a result, we are now in the process of incorporating LiDeC into the issue tracking system of the company.

6.6 FUTURE WORK

The next step in our research concerns analysis recipes—guidelines on how to analyse the collected data. We envision two distinct types of analysis recipes: post-mortem and in-process—post-mortem recipes will mainly be support for organizational learning, whereas in-process would serve as a tool for project control.

Post-mortem recipes would concentrate on analysing the collected data in order to learn about a finished project; e.g. evaluating whether changes in the way of working in the project yielded any noticeable effects, or analysing weak spots in the way of working as promising candidates for future improvements.

In contrast, in-process recipes would be guidelines on how to use collected defect data from previous project phases in order to predict future phases within the project. As part of this we will need to identify relevant predictors. The challenges include the absence of source-code predictors; instead there is a need to identify predictors based on specification and design artefacts, e.g. requirements and design model complexity.

By establishing a defect profile baseline per development phase, we aim at developing a way to predict future defect inflow. Such a prediction model would, for instance, assist in resource planning in a similar way as presented by Bijsma et al. (Bijlsma et al., 2011) where mainly source-code related predictors were used to estimate the time it would take to resolve a defect—a metric that can be used to assist in prioritizing defects.

6.7 ACKNOWLEDGEMENTS

This research is partially sponsored by The Swedish Governmental Agency for Innovative Systems (VINNOVA) under the Intelligent Vehicle Safety Systems (IVSS) programme. In addition, the authors would like to thank all the people at VCC who participated in the study, and people who generously took the time to review this manuscript.

IV

PART IV—EVALUATION

The only man who behaves sensibly is my tailor; he takes my measurements anew every time he sees me, while all the rest go on with their old measurements and expect me to fit them
— George Bernhard Shaw

7 AN INITIAL EVALUATION OF THE PRACTICAL FEASIBILITY OF LIDEC

The chapter presents an initial industrial evaluation of LiDeC. The chapter provides initial validation of the practical viability of the adapted classification scheme. The chapter[11] was previously published as:

> Mellegård, N., Staron, M., Törner, F.
> *A Light-weight Defect Classification Scheme for Embedded Automotive Software and its Initial Evaluation*
> Published at 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE) 2012

7.1 INTRODUCTION

Software reliability is of central importance in modern cars as software controlled systems are becoming increasingly pro-active—recent safety

[11] In this chapter, the background section has been removed as it was also published as part of the study reported in chapter 6.

systems are, for instance, able to automatically apply brakes to avoid or mitigate the effects of a crash. The car manufacturers (Original Equipment Manufacturer, OEMs) need, in order to achieve reliability, effective ways to manage defects during development (in-process) and during run-time (e.g. fault tolerance mechanisms). For the in-process defects it is important to identify, analyse and remove defects which could compromise the reliability of the cars. Furthermore, identifying patterns in the in-process defects enables effective detection and removal of defects, for instance by indicating which test activities to focus on, or supporting fault injection strategies. Such patterns may also serve as a maturity assessment metric for the software under development and as a prediction model of defect inflows in future project phases. In order to identify such patterns, however, systematic and structured defect documentation is required.

Defect documentation and analysis is common practice in most software development organizations. Its benefits are further emphasised through the inclusion in process maturity models—such as CMMI (CMMI Product Team, 2010a) and SPICE (The SPICE User Group, 2011)—as they require systematic defect documentation, analysis and follow-up. Neither CMMI nor SPICE, however, specifies how such defect documentation and analysis is to be done. Companies thus have their own interpretations resulting in varying quality of defect documentation; for instance, ambiguous interpretation of data or subjective opinions of the reporter. In addition, the automotive industry is implementing the ISO 26262 standard (ISO/DIS, 2011) for assuring functional safety, where functional safety assessment is one important confirmation measure of achieved safety. Defect reports are part of the overall safety documentation and are as such included in the base for the safety assessment. Hence, there is a need for a structured approach to defect documentation.

There have been several approaches proposed on how to perform structured collection and analysis of defect information; e.g. defect taxonomies (Beizer, 1990), root cause analysis (RCA) (Leszak et al., 2000) as well as various defect classification schemes such as Orthogonal Defect Classification (ODC) (Chillarege et al., 1992), the HP scheme (Grady, 1992) and IEEE Std. 1044 (IEEE, 2009). Although shown to be useful these approaches were designed for specific contexts (Freimut et al., 2005) or may be too generic to be directly applicable, causing the need for adaptations (Wagner, 2008); such adaptations have been identified as one of the major challenges in applying a defect classification scheme (Freimut et al., 2005; Wagner, 2008). Specifically, defect classification approaches often assume full knowledge of the defects, i.e. have a source-code focus and presuppose

ownership of the software components (Freimut et al., 2005). This poses challenges for organizations where software is developed by suppliers—a situation common for the development of software systems in the automotive domain: even though software components (e.g. ABS or collision warning system) are often developed by suppliers, the quality of the complete product—the car—is the responsibility of the OEM. This entails the need to systematically analyse and follow-up on the quality of the supplied software components.

Furthermore, defect documentation—however important—may be seen as a mainly administrative task that does not directly contribute to the end-product. Thus, the defect documentation approach taken should require a minimum of analysis effort in addition to what is needed to identify and remove the defect, while still providing the additional benefit of characterizing the quality of the product and development process (Freimut, 2001).

In this paper we address these challenges by investigating how a defect classification scheme can be adapted to the automotive software development context by studying the development of active safety features. More specifically, the aims of our adapted classification scheme—the Light-weight Defect Classification scheme (LiDeC)—include:

- To be useful in a development context with limited insight into source-code
- To minimize the learning and classification time required

The LiDeC scheme was developed as part of a case study at Volvo Car Corporation (VCC[12]) and initially evaluated with a sample of problem reports from a project finished a year prior to the study. The results from the initial evaluation showed that developers quickly learnt to apply the classification scheme, and that the required time to classify a defect was substantially lower than with other approaches to defect documentation. Moreover, we were, from initial analyses of the defect data, able to discover patterns in the defects that were contrary to what was anticipated by the development teams. These findings may contribute to future improvements to the development practice.

The rest of the chapter is structured as follows: section 7.2 describes the method used, section 7.3 summarizes the results and the final sections conclude the chapter and outlines future work. For background and context, see section 6.2.

[12] http://www.volvocars.com

7.2 METHOD

The research presented in this paper followed the case study method described by Yin (Yin, 2009). The case study method was considered appropriate as the applicability of the adapted classification scheme was of importance; adapting the IEEE Std. 1044 in the same context as it will be deployed would increase the chances of it being useful in that context. Specifically, we have used a single-case design by applying the classification scheme to defect reports from one project at our case company. The rationale for this was that the defect inflow profile from the project (shown in Figure 14) was considered representative by the developers—similar inflow profiles had been observed in other projects. Using the defect reports from the project, we developed an adapted defect classification scheme; in particular, we addressed the following research questions:

E1 *How can the IEEE Std. 1044 be adapted to a software development context with limited access to the source-code?*

E2 *How can the IEEE Std. 1044 be adapted to minimize its process foot-print in terms of required learning and classification time?*

As a practical guideline to adapting a defect classification scheme we followed Freimut (Freimut, 2001). The study was conducted in the following three stages:

Stage 1: Establish terminology. The aim of the first stage of the study was to establish a set of classification attributes using terminology aligned with the case company.

As a base for developing the classification scheme we used the IEEE Std. 1044 (IEEE, 2009) and its guide (IEEE, 1996). In this stage we began by choosing attributes from the IEEE Std. 1044 that we—based on our previous research (Mellegård and Staron, 2010a, 2010b, 2010c)—found relevant for the specific development context at the company; for instance attributes related to customer value or societal impact was not considered relevant for the development phase under study.

During two one hour-long interviews, we explained the initial classification attributes to the interviewee (project leader), and asked the interviewee to relate these attributes to the case company. We took notes during these interviews and refined the classification scheme according to these notes.

Stage 2: Tune feasibility. The second stage of the case study aimed at streamlining the set of values each attribute could be assigned.

The stage consisted of a two hour long interview with a developer in which a number of defects were classified. The set of values for each attribute were evaluated during the classification session: (a) if an attribute value was never assigned and the interviewee could not give an example of when the attribute value would be given, it was considered for removal, and; (b) if a defect was to be classified according to an attribute and the interviewee did not consider any of the available attribute values sufficiently described the defect, a new value was considered for addition.

From the results of the second stage the final classification scheme was compiled and depicted in the form of flowcharts with short questions providing a guide to arrive at the correct attribute value. Each attribute value was, furthermore, provided with a short illustrative example.

Stage 3: Scheme evaluation. The final stage of the study aimed at evaluating the efficiency and effectiveness of the classification scheme.

In this stage defect reports were classified according to the scheme. Four subjects involved in the project participated in six separate two hour long classification sessions (two subjects participated in two consecutive classification sessions). Three of the subjects were not involved in the previous two stages of the project.

The project used as case had finished one year prior to the study and contained over 100 problem reports[13]. All subjects involved in the study had been part of the project with the following roles: two developers, one tester and one project leader.

In the third stage of the study we were able to classify 22 defects selected randomly using blocking; the defect reports were divided into two blocks, where the peak close to software release (shown in Figure 14 on page 69) was assigned as one block and the remainder of the project as the second block.

The results included measuring:

- *Learning time*– we measured the time taken to classify each defect. Since we explained the classification scheme while doing the actual classification, we could measure the improvement in time per defect and use this as a metric to estimate required time to learn the classification scheme;

[13] Exact number cannot be disclosed due to confidentiality reasons

- *Efficiency* – Efficiency was considered the time required per defect once the scheme was known by the subject. Because the project had been finished a year prior—and that the defect reports spanned a number of years—we consider this to be a worst-case scenario; if the defect classification is done in-process instead of post-factum, we believe that the time required to classify a defect should be equal or less than in this study;
- *Repeatability*– Four randomly selected defects were classified by all four subjects. The results were compared in order to evaluate the repeatability of the scheme;
- *Effectiveness*– In order for the classification scheme to be effective we must be able to analyse the collected data and use it learn about the process. As we lowered the level of detail of the data captured by the classification—compared to e.g. ODC or IEEE Std. 1044—there was a risk that too much details were removed for the data to lend itself to meaningful analysis. In order to evaluate the effectiveness of the scheme we stratified the space of defects and chose a random sample from one stratum: the spike in defect inflow after one late test phase of the project (see Figure 14). Although the developers did anticipate reasons for the spike in defect inflow, getting more formal evidence about these defects was considered interesting by the case company.

7.2.1 Validity Evaluation

We have identified and grouped the threats to validity in our study according to recommendations of Yin (Yin, 2009):

- *Construct validity* – By basing our defect classification scheme on the IEEE Std. 1044 and by keeping careful notes on how to map concepts specific to our case to the standard, we consider that the threat to construct validity to have been minimized. Furthermore, as both the adaptation of the classification schemes and the evaluation was done using real defect data from an industrial project with the assistance of the developers involved in the project, we consider the threat to construct validity to have been further reduced.
- *Internal validity* – As any interview study we anticipated some personal bias in the answers from the interviewees. In order to minimize this threat we triangulated the results by including multiple subjects in our interviews. In addition, a set of defects were classified by multiple subjects thereby allowing evaluation of the repeatability of

the classification scheme; section 7.3.2 reports the results from this evaluation.

- *External validity* – There is a risk that the results are too specific to Volvo Car Corporation. However, as we documented and justified the modifications done to the IEEE Std. 1044 as well as described the particular development context of our case, we believe that our results can be generalized to similar contexts outside our specific case. Moreover, we consider the mapping between attributes of the classification schemes provided by Freimut (Freimut, 2001) to contribute to the generalizability of our classification scheme; e.g. the mapping between classification schemes enables analysis methods utilized with other schemes to be applicable to LiDeC as well, and therefore we believe that results are also comparable.
- *Reliability* – As part of the case study design, we have created a case study protocol which ensured that we conducted the study and collected the data in a consistent manner. By using this protocol, we believe that the study can be reliably reproduced.

7.3 RESULTS

As described in section 7.2 we aimed at designing a classification scheme, based on the IEEE Std. 1044, adapted to the development of automotive safety features, while being light-weight with respect to learning and classification time. The main challenge of adapting the IEEE Std. 1044 included tailoring the attributes relating to the fault and its resolution; specifically attributes in the phases Investigation, Action and, Impact Identification. Furthermore, the classification scheme was validated with respect to its required process foot-print and the usefulness of the resulting data.

The results are reported below in two parts; first, the classification scheme is presented, and then its initial evaluation is presented.

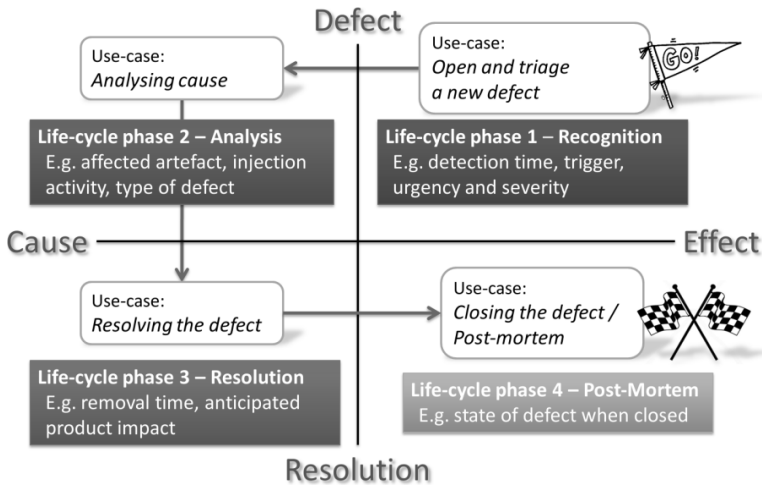

Figure 16. Overview of the LiDeC Scheme

7.3.1 The LiDeC Scheme

The LiDeC scheme captures—as shown in Figure 16—attributes from four phases of the defect life-cycle (IEEE, 1996, chap. 8.1) (described in more detail below): the first phase captures information about the recognition of the defect, i.e. observing a deviation (failure); the second phase captures information about the underlying cause of the defect (referred to as Investigation in IEEE Std. 1044.1 (IEEE, 1996)); in phase three information about the defect resolution is captured (referred to as Action in IEEE Std. 1044.1 (IEEE, 1996)); and the last phase captures information about what was actually done about the defect (referred to as Disposition in IEEE Std. 1044.1 (IEEE, 1996). Table 10 shows a comparison between the life-cycle phases of IEEE Std. 1044, ODC and LiDeC schemes.

Table 10 Mapping of defect life-cycle phases

Defect life-cycle phase	*IEEE 1044*	*ODC*	*LiDeC*
1	Recognition	Open	Recognition
2	Investigation		Analysis
3	Action	Close	Resolution
4	Impact Identification		
5	Disposition		Post-mortem

As can be seen by the number of attributes in each phase (described in detail in the subsections below), the main focus of the LiDeC scheme is on recognition and analysis of the defect. The justification is that the later a defect is discovered the more costly its resolution tend to be (Boehm, 1981). Consequently, the main focus of the classification scheme is on how defects are discovered, how the product is affected by them, and what types of defects they are. Analysing this information will contribute to understanding which phases of the development process contain the most improvement potential.

The phases of the classification scheme are aligned with the defect management process at the case company and directly correspond to the states a defect can be assigned: recognized, analysed, resolution proposed and post-mortem. The following sections describe the attributes of each phase.

Recognition

The attributes in the first phase of the defect life-cycle (shown in Table 11) relate to data about the discovery of a failure and its effects on the system in question, i.e. the manifestation of the defect. The attributes capture project related information—timing of detection and how urgently the defect needs to be removed—and product related information—end-user perceived severity of the defect and how the defect affect the product.

This information will be used in analysis, for example, to assess whether the most serious defects are detected timely, investigate which activities are more effective in detecting the most severe defects.

Table 11 Classification attributes for the Recognition phase

Attribute	*Question*	*Values*
Timing / Detection	When was the defect detected?	Date and project phase
Timing / Preferred	When should the defect have been detected (subjective)?	Project phase if different from Timing / Detection
Affects S/W	Does the defect affect software?	Yes / No
Detection Activity	What was done to detect the defect?	Inspection/Requirements, Inspection/Design, Unit test/In-house, Unit test/Supplier, System test/bench, Functional test/Test track, Functional test/Expedition Production/Manufacturing Production/Customer report
Urgency	How urgently does the defect need to be addressed?	Immediately, Next development release, Before start of production, Deferrable
Severity	How severe is the defect with respect to product quality?	None, Nuisance, Limited Functionality, Show-stopper
Effect	How does the defect primarily affect the product?	Capability/Undesired activation, Capability/Inactive on true positive, Capability/Other, Function Safety, Maintainability, Usability, Testability, Configurability

Analysis

The attributes in the second phase of the defect life-cycle (shown in Table 12) aim at capturing data about the underlying fault causing the failure; e.g. in

what work product and product component the problem originated, when in the process the problem was injected and the type of defect.

Table 12 Classification attributes for the Analysis phase

Attribute	*Question*	*Values*
Artefact	Which software work product contained the defect?	Req./Internal, Req./Cross-function, Req./External, Design model, Impl./Executable model, Impl./Code, Impl./Configuration params, Tool
Injection activity	When was the defect injected?	Specification, Design, Impl./In-house modelling, Impl./Suppl. mdl. transform., Impl./Supplier coding, Configuration
Component / Asset	Which design component contained the defect?	Internal (product) module name also identifying its version
Type	What type of defect was it?	Description, Data, Interface / Timing, Logic / Algorithm, Tooling, Tuning

Resolution

The attributes in the third phase of the defect life-cycle phase (shown in Table 13) aim at capturing data about the proposed resolution. As implementation specific details of the resolution may not be available, the attributes in this phase focus on capturing the cost of resolving the defect in terms of development effort. Specifically, to capture what impact a resolution would have on the product and on the process; the impact on the product is captured in terms of how much of the product would be affected by the modification, and impact on the process in terms of amount of regression testing needed.

Table 13 Classification attributes for the Resolution phase

Attribute	*Question*	*Values*
Removal time	When was the defect report closed?	Date and project phase
Product impact	What would the impact of a proper resolution be on the product?	None, Local (unit) modification, Multiple components, Funct. changes (re-design)
Required Verification Level	What level of regression testing would a proper resolution require?	None, Inspection, Unit test, System test, Expedition

Furthermore, the attributes of the Resolution phase are intended to capture data about a proper resolution of the defect; in practice, one could choose to resolve defects by means of workarounds (this data is captured in the final phase of the classification scheme).

Post-mortem

In the last phase of the defect life-cycle the single attribute (shown in Table 14) record what was finally done to close the defect; to what extent the defect was resolved.

Table 14 Classification attributes for the Post-mortem phase

Attribute	*Question*	*Values*
Resolution state	What was the final state of the defect when the problem report was closed?	Corrected (proper resolution applied), Workaround/Fix, Workaround/Product de-scoped, No Action/Deferred, No Action/Referred, No Action/Not found, No Action/No action

7.3.2 Evaluation—Efficiency and Effectiveness

This section reports on the initial evaluation of the classification scheme. The evaluation was made based on efficiency and effectiveness. Efficiency was measured in terms of the time required to learn the classification scheme, and the time required to classify a defect. Effectiveness, on the other hand, was measured in terms of classification repeatability and what conclusions can be drawn from analysing the classification data.

Efficiency—Learning time and Classification Rate

In the third stage of the study, the time required to classify each defect was measured in order to evaluate learning time and classification rate; Figure 17 shows the time series for the two subjects that participated in the two consecutive classification sessions. As can be seen in the chart, the first $2-4$ defects in the first session took substantially longer to classify, but after that the time remained stable. Furthermore, the learning time required in the second session was substantially lower (there was approximately one week between sessions for both subjects). We consider this as an indication that the scheme is understandable and quick to learn.

In addition, once the classification scheme was understood by the subject, the time required to classify a defect remained stable around $5-10$ minutes. Considering that the project had been finished one year prior to the study, and that some of the defect reports included were several years old, we believe that this may be considered a worst-case scenario with respect to classification time.

When asked, all subjects said that the information captured by the classification scheme would be known at the time of reporting the defect. Consequently, if applied in-process, classifying according to the LiDeC scheme would require no additional analysis effort; the scheme would thus capture tacit knowledge and could therefore be considered cheap in terms of project resources.

Effectiveness—Repeatability

To provide initial data to evaluate the repeatability of the classification scheme four randomly selected defects were classified by all subjects in the final stage of the study. Differences were found in 6 of the 15 attributes, specifically: *Detection activity*, *Severity*, *Effect*, *Type*, *Required Verification Level* and *Resolution State*. These differences can in part be attributed to the fact that the classification was done based on sparsely documented defect reports.

However, we found that illustrative examples for each attribute value would contribute to the repeatability. For instance, in one case for the attribute *Resolution State*, where the defect report indicated that a defect had already been resolved in an earlier release of the software, one subject classified the defect as *Corrected* (as the defect had been properly resolved) while another subject chose *No Action* (as nothing had been done in response to that particular defect report).

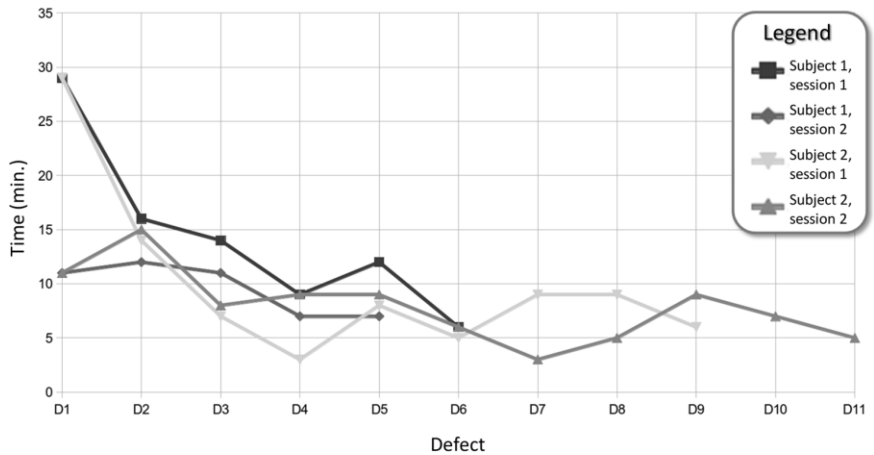

Figure 17. Classification time per defect

Effectiveness—Data Analysis

In order to evaluate the trade-off between classification effort required and the analysis power of the data, an initial data analysis was done. The project from which the defect reports were used had an inflow spike in a late project stage (as shown in Figure 14 on page 69)—similar spikes had been observed in both previous and subsequent projects. The hypothesis posed by the development teams was that the defects were related to integration issues—the spike in defects correlated with the first test vehicle built entirely with components intended for production. In the final stage of the case study 8% of the defects from the inflow spike were classified with the intention to evaluate the hypothesis.

The results showed that—contrary to the hypothesis of the teams, and also contrary to the results of our previous studies (Mellegård and Staron, 2010a, 2010b, 2010c)—the predominant defect type reported was related to

algorithms and code logic. Furthermore, the results indicated that these late issues—in the opinion of the subjects—should have been detected on unit or system testing level.

The predominance of algorithm and code logic defects in the classified data was explained by the domain expert and was logical for the studied organization and development process. Improvements based on this explanation were also identified by the expert, which indicates that the defect classification scheme forms a useful contribution to the efficiency of software development at the company.

7.4 CONCLUSION

In this chapter we have examined how efficiency of defect classification can be improved in a development context where code is developed by suppliers and the full knowledge about the source code is linked with intellectual property rights. Specifically, we have adapted the IEEE Std. 1044 to minimize its process foot-print in terms of required learning and classification time. We also evaluated the adapted classification scheme—the LiDeC scheme—at Volvo Car Corporation. The adaptation consisted mainly of raising the abstraction level of the attributes related to source-code and to the resolution of the defect. Focus was instead shifted to the recognition phase to enable evaluation of the efficiency of detection activities.

Lowering the granularity of the attributes may, however, risk compromising the usefulness of the results of the classification; a risk that was assessed by conducting an initial evaluation of the collected data. This initial evaluation showed that even though the granularity of the attributes were lower, analyses of the data collected were still able to contribute with new information about the development process; for instance, whereas the typical type of late defects were anticipated to be integration issues, such as timing, the classification indicated the it was instead algorithms and code logic defects.

Furthermore, the attributes selected in the LiDeC scheme were considered by the subjects of the study to be knowledge already possessed by the developers when reporting the defect; thus capturing tacit knowledge.

As a result of this initial study, we are currently in the process of incorporating the classification scheme in the company's defect reporting system.

7.5 FUTURE WORK

The next step in our research concerns analysis recipes—guidelines on how to analyse the collected data. We envision two distinct types of analysis recipes: post-mortem and in-process. Post-mortem recipes will mainly be support for organizational learning, whereas in-process would serve as a tool for project control.

Post-mortem recipes would concentrate on analysing the collected data in order to learn about a finished project; e.g. evaluating whether changes in the way of working in the project yielded any noticeable effects, or analysing weak spots in the way of working as promising candidates for future improvements.

In contrast, in-process recipes would be guidelines on how to use collected defect data from previous project phases in order to predict future phases within the project. As part of this we will need to identify relevant predictors. The challenges include the absence of source-code predictors; instead there is a need to identify predictors based on specification and design artefacts, e.g. requirements and design model complexity.

By establishing a defect profile baseline per development phase, we aim at developing a way to predict future defect inflow. Such a prediction model would, for instance, assist in resource planning in a similar way as presented by Bijsma et al. (Bijlsma et al., 2011) where mainly source-code related predictors were used to estimate the time it would take to resolve a defect—a metric that can be used to assist in prioritizing defects.

7.6 ACKNOWLEDGEMENTS

This research is partially sponsored by The Swedish Governmental Agency for Innovative Systems (VINNOVA) under the Intelligent Vehicle Safety Systems (IVSS) programme. In addition, the authors would like to thank all the people at VCC who participated in the study.

8 A COMPREHENSIVE EVALUATION OF DEFECT CLASSIFICATION SCHEMES

The chapter presents a comprehensive evaluation of LiDeC, consisting of a controlled experiment and an industrial case study. In addition to evaluating LiDeC, the chapter provides a detailed description and reflection of the evaluation methodology. In doing so, the chapter is intended to propose an established process to assess efficiency and effectiveness of defect classification schemes. The chapter has been submitted for publication as:

> Mellegård, N., Staron, M., Törner, F.
> *Evaluation of Defect Classification Schemes — An Experiment*
> Submitted to Software Quality Journal in 2013 (Ref. no: SQJO829)

8.1 INTRODUCTION

Reliable and relevant measurements are essential in order to manage a complex software development process. It is through measurements that, for instance, the quality of products or the efficiency of development processes can be characterized, evaluated, predicted and improved (SEI, 2013). One source of measurements is defects reported during development and after deployment. Defect reports can contain much interesting data; such as, how the defects were detected, what type they were and what action was required to resolve them. When analysed quantitatively, patterns in the defect reports may be discovered that, for example, could provide information about the effectiveness of various test activities or facilitate prediction of product quality. Defect reports, however, are often written in free text, making quantitative analyses difficult. Defect classification schemes address this problem, by providing defect reports with a shared structure, and thus enabling more efficient data collection.

Even though the concept of DCS has, since the introduction of ODC (Orthogonal Defect Classification) (Chillarege et al., 1992) and IEEE Std. 1044 (IEEE, 1993, 2009), been around for more than two decades, it has to our best knowledge (Mellegård et al., 2013) received little research attention. In particular, while it has been recognized that a major challenge in applying

a DCS in an organization is to adapt it to suit the particular development context (Freimut, 2001), it is not clear how to evaluate the adapted scheme with respect to its efficiency (time required to learn and apply the DCS) and effectiveness (quality and usefulness of the classification data).

In this paper, we report on a two-part evaluation of LiDeC (Light-weight Defect Classification scheme) (Mellegård et al., 2012a, 2012b), a DCS adapted to the development of automotive active safety features. The two parts of the evaluation aim at examining aspects of the two main phases of the defect classification approach, illustrated in Figure 18. Denoted by *A* in Figure 18, initially defect reports, often written in free text, are classified according to a classification scheme. The classification data can then be analysed (denoted by *B* in the figure) to discover patterns, which may reveal information about quality of the products or efficiency of the development process.

The first part of the evaluation—aimed at examining aspects of LiDeC as a data collection tool (as denoted by *A* in Figure 18)—was conducted as a controlled experiment using university students as subjects. In the experiment, the quality of classifications done using LiDeC was compared to classification done using ODC. The results were analysed in terms of classification time, accuracy and consistency. In the second part of the evaluation, the focus was on the analysing the classification data collected using LiDeC (denoted by *B* in Figure 18). The evaluation was conducted as an industrial case study at Volvo Car Corporation (VCC) in which a sample of defects from a finished project was classified. The main objective of the case study was to evaluate how the data collected using LiDeC can provide useful information in an industrial context.

This paper is intended for both practitioners and researchers. For practitioners, specifically with an interest in software quality management, the paper provides an evaluation of DCS as a data collection tool. In the evaluation is included the cost of using a DCS in terms of classification time, and the reliability of the data collected. In addition, the potential benefit of the data is examined through an industrial case study with the aim to characterize what insights analyses of the data can bring. For researchers, the paper describes a rigorous and comprehensive method for evaluating DCSs. Although evaluations of specific aspects of a DCS have been reported previously, this paper contributes with a more comprehensive methodology for evaluating both efficiency and effectiveness. Such evaluations are important, as DCS data, for instance, is often used to demonstrate effects of applying new techniques in software development—we elaborate on this in

the following section. In order to use DCS data as means of such validation, it must be shown to be reliable.

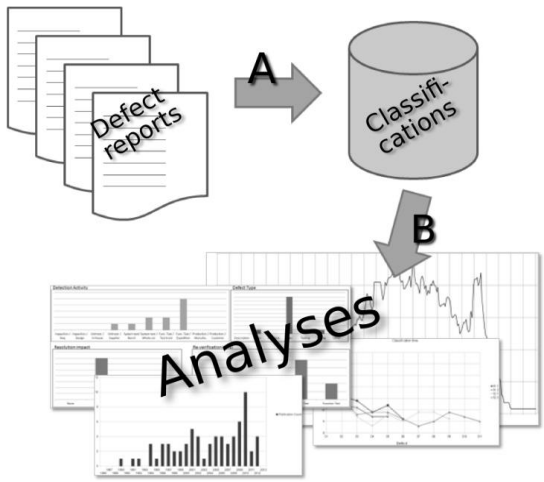

Figure 18. Defect classification, and the focus of the current study. The focus of the current study is on the two processes A and B, where A denotes the process of applying a classification scheme to defect reports, and B denotes analyses of the classification data

The remainder of the chapter is structured as follows. Section 8.2 presents a selection of work related to the present article. Section 8.3 outlines defect classification schemes in general, and ODC and LiDeC in particular. Section 8.4 gives a detailed methodological description, and section 8.5 presents the results. Section 8.6 highlights and discusses the main findings and section 8.7 concludes the chapter.

8.2 Related Work

The importance of establishing a reliable system of process and product metrics is demonstrated by its inclusion in process maturity models, such as CMMI (CMMI Product Team, 2010b) and Automotive Spice (The SPICE User Group, 2011). In these models, a prerequisite for advancing in levels of maturity is that aspects of the products and their development process are measured, and defects detected during development and after deployment is one source of such measurements.

There are numerous approaches to acquire data from defects. While approaches such as post-mortem reviews (Dingsøyr, 2005) and root-cause

analysis can provide valuable information about process issues and their underlying cause, they are qualitative, rather than quantitative as would be required in higher levels of process maturity in the models. To obtain quantitative data from defects, fault taxonomies (e.g. (Beizer, 1990)) aim at classifying the type of defect. The type of fault, however, is merely one aspect of a defect. Defect classification schemes aim at capturing data from multiple aspects, where defect type is one such aspect. There have been a number of DCSs proposed; for instance, ODC (Chillarege et al., 1992), HP (Grady, 1992), LiDeC (Mellegård et al., 2012b), and an adaptation of ISO/IEC 9126 (Vetro' et al., 2012). While IEEE Std. 1044 aims at standardizing the structure of a DCS, there have been few reports on its applicability (Freimut, 2001). Instead, ODC seems to be the most widely used (Vetro' et al., 2012) and reported on in literature. In his paper, Freimut (2001) provides an overview of defect classification schemes in general, and ODC, HP and IEEE Std. 1044 specifically.

Freimut, furthermore, identifies that a major challenge in applying a DCS in an organization is to adapt it to the specific development context of that organization. Research on such adaptation has, for instance, been reported by Li et al. (2010b) in which ODC was adapted specifically to black-box testing. In our previous work, we have reported on the adaptation of IEEE Std. 1044 to the development of automotive active safety software (Mellegård et al., 2013, 2012a). Other approaches to improving DCS include automating the classification, fully (Thung et al., 2012) or partially (Huang et al., 2011; Wang He et al., 2009). In their paper, Vetro et al. (2012) examined one attribute from ODC specifically (the *Impact* attribute) and redefined it according to the ISO/IEC 9126 standard for software product quality (ISO/IEC, 2001). The adaptation results, according to the authors, in a more comprehensive definition of the attribute. Additionally, the adapted attribute is hierarchical, with a main characteristic (e.g. maintainability), and a number of sub-characteristics (e.g. analysability, stability, testability).

Evaluations of a DCS—the focus of the present study—have been conducted, both in academy and in industry. In their paper, El-Emam and Wieczorek (1998) conducted a controlled experiment that aimed at examining the repeatability of classification. In their experiment, El-Emam and Wieczorek examined agreement between two subjects (practitioners) classifying the *Type* attribute as defined by ODC, for 605 defects. The results of the experiment were analysed using Cohen's kappa (Landis and Koch, 1977; Sim and Wright, 2005) and showed a high degree of agreement (a replication by Freimut (2005) showed similar results). Henningsson and Wohlin (2004) conducted a similar experiment in which eight student

subjects classified the *Type* attribute for 30 defects. In contrast to El-Emam and Wieczorek, the results, also analysed using Cohen's kappa, showed low classification agreement among the subjects. A third study on the repeatability of classification was conducted by Vetro et al. (2012) on their proposed adaptation of the ODC *Impact* attribute. In their experiment, six subjects classified the adapted attribute for 78 defects. Three of the subjects were students and three were researchers (non-practitioners). While the results, analysed using Cohen's kappa, showed fair agreement among the subjects (according to the scale presented by Lanis and Koch (1977)) for the main attribute characteristic, on the more detailed sub-characteristic level the agreement was considerably lower. While these evaluations, conducted in an artificial setting, provide evidence on the efficiency of certain aspects of DCSs, there are, to our best knowledge, no comprehensive studies on applying classification schemes—as is the aim of the present study.

Evaluations on the industrial applicability of DCSs have been reported, for instance by Butcher et al. (2002), Chillarege and Prasad (2002), Freimut (2005), and Li et al. (2012). Butcher et al. conducted case studies where ODC was applied at three software development organizations. In all three cases, the classification was found to contribute to improving the practice; for instance, the pre-release test effectiveness was improved by identifying the typical triggers for defects that escaped into the field. All three companies, furthermore, decided to continue the defect classification practice. Li et al., conducted two-round case studies at two companies. In the first round, they analysed defect classification data from projects to identify process improvement opportunities. In the second round, conducted 6 and 12 months later, classification data was analysed to evaluate whether effects of the improvement initiatives could be found. At both companies, expected effects were indeed found. These studies show that defect classification data can contribute to characterize a process, to identify improvement opportunities, and to evaluate the effects of changes to that process.

The academic value of defect classification data has been demonstrated, for instance by Nagappan et al. (2004), Zheng et al. (2006) and, Jin and Jiang (2009). In the papers by Nagappan et al. and Zheng et al., the effects of applying automated software inspection techniques analyses were examined by using defect classification data. Specifically, the distribution of defect types was compared between projects that used inspection techniques and projects that did not. In both paper, the authors were able to demonstrate specific and statistically significant differences in the distribution of the defect types, thus providing evidence of the effect of using automated inspection techniques. In their paper, Jin and Jiang used ODC data to develop

a fault injection scheme for embedded systems. Specifically, they examined the mapping of faults in a high-level language (C code) to their mutation in machine code. The purpose was to evaluate fault-injection on machine code level, to be used when source code is not available. These studies show that defect classification data can be used to demonstrate effects of applying various techniques—a common theme in much software engineering research. This, however, presumes that the data produced by a classification scheme is reliable; something that has not been shown conclusively in the current body of knowledge.

8.3 DEFECT CLASSIFICATION

In any larger software organization, many factors add complexity to the development process. To manage a complex software development process, effective measurement practices are needed; defects detected during development and after deployment is one source of such measurements. Defects, although expected during development, can be seen as a symptom of weaknesses in the process—especially if defects, for instance, tend to slip through the test activities that are meant to detect them.

One method of collecting data about defects is DCSs. A DCS intends to provide defect reports with a shared and formal structure by defining a set of attributes, and for each attribute a set of values that can be assigned. Each attribute captures data about one aspect of the defect – for instance, in which type of test the defect was discovered. Typically, the aspects of a defect can be divided into the major life-cycle phases: *Detection* and *Resolution*. The detection phase consists of identifying a problem (e.g. a failure) and analysing what caused it (i.e. the underlying fault), while the resolution phase consist of identifying a suitable action to remove either the fault or its manifestation.

In the first part of our study, the efficiency and effectiveness of two classification schemes are examined; ODC and LiDeC. In the following two subsections each of these two schemes are described in more detail.

8.3.1 ODC

In ODC the attributes are organized into the two life-cycle phases *Open* and *Close*. As can be seen in Table 15, showing an overview of ODC, the attributes in the *Open* phase focus on the detection of the defect, whereas in the *Close* phase the attributes focus on aspects of the fault and its resolution.

Table 15 Overview of ODC (adapted from (Freimut 2001))

Phase	*Attribute*	*Meaning*
Open	Activity	When did you detect the defect?
	Trigger	How did you detect the defect?
	Impact	What would the customer have noticed if the defect had escaped into the field?
Close	Target	What high-level entity was fixed?
	Source	Who developed the target?
	Age	What is the history of the target?
	Type	What had to be fixed?
	Qualifier	Was the defect caused by something missing, incorrect or extraneous?

In the *Open* phase, the *Activity* and *Trigger* attributes relate to the test procedures that were used to detect the defect. The values available for the *Trigger* attribute are dependent on the value chosen for the *Activity* attribute; for instance if the value *Unit test* is chosen as the detection activity, then only the *Trigger* values relevant for unit tests are available (i.e *Simple path* and *Complex path*). The third and final attribute of the *Open* phase, *Impact*, captures how the customer would have perceived the defect if it had remained in the system—e.g. reliability, usability, capability etc.). The attributes in the *Open* phase capture data about the black-box properties of the defect. In the following phase, focus is shifted to aspects of the underlying fault and its resolution.

In the *Close* phase, the attributes *Target*, *Source* and *Age* relate to aspects of the work product that was modified in order to resolve the defect. These attributes provide data about which artefact contained the fault (e.g. requirements, design, code, etc.), who was responsible for developing it (e.g. in-house, outsourced, etc.) and whether it was the creation or a modification of the artefact that caused the defect. The attributes *Type* and *Qualifier* are related to the aspects of the fault. Type aims at capturing data about the (mainly source code related) nature of the fault (e.g. assignment or checking), and whether something was missing, incorrect or extraneous (e.g. a missing variable assignment).

8.3.2 LiDeC

In LiDeC the attributes are divided into four separate phases (illustrated in Figure 19):

- The *Recognition* phase in which an issue is identified;
- The *Analysis* phase in which the underlying cause of the issue is identified;
- The *Resolution* phase in which the various possible modifications to rectify the issue are developed, and finally;
- The *Post-mortem* phase in which the actual action taken is analysed.

In the recognition phase, shown in the top right of Figure 19, a failure is identified and attributes related to its manifestation are captured: what was done to detect the failure (*Detection activity*, corresponding to *Activity* in ODC), the estimated urgency of finding a solution (*Urgency*) and the estimated impact on the product (*Severity*, *Effect* and *Functional Safety Impact*).

In the following phase, shown in the top left of Figure 19, the cause of the failure (i.e. fault) is identified and attributes related to the fault are captured. These attributes include: which software work product contained the defect (*Artefact*, corresponding to the ODC attribute *Target*), e.g. requirement, design component or code; what activity caused the fault to occur in the artefact (*Injection activity*); and finally what the type of defect was (*Type*).

The third life-cycle phase, shown in the bottom left of Figure 19, capture data about the proposed resolution of the defect in terms of its impact on the product (*Resolution Impact*)—specifically, to what extent the product needs to change—and on the process in terms of the type of testing needed to verify the resolution (*Required Verification Level*).

In the final life-cycle phase, shown in the bottom right of Figure 19, the single attribute (*Resolution State*) capture what was actually done to close. Although there may be an action proposed on how to resolve the defect properly, in practice workarounds can be an alternative to applying a proper resolution (i.e. removing the manifestation of the fault, rather than the fault itself).

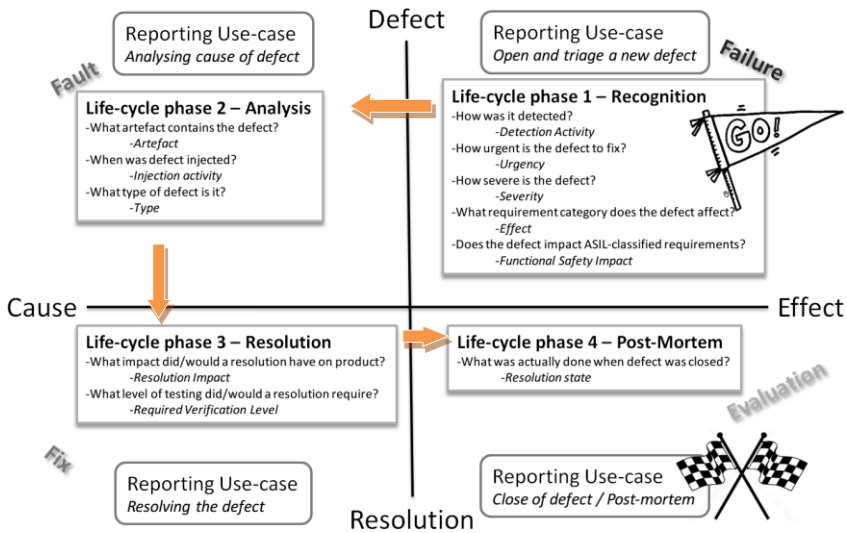

Figure 19 LiDeC overview

8.4 METHODOLOGY

The evaluation aimed at examining LiDeC from two perspectives, as shown in Figure 18: applying the classification scheme (denoted by *A* in the figure), and analysing the classification data (denoted by *B* in the figure). The first part of the evaluation was conducted as a controlled experiment, while the second as an industrial case study. The methodologies of the two studies are described in the subsections below.

8.4.1 Part I: Experiment

The aim of the experiment was to examine whether applying a classification scheme adapted to a specific context (LiDeC) have significant advantages or disadvantages compared to a generic classification scheme (ODC). For this purpose, a single factor, two-treatment experiment design was applied. The factor in the experiment was the defect classification scheme (LiDeC and ODC used as the treatments). In the experiment, two groups of subjects classified five shared defects according to one of the two classification schemes. The comparison between the two schemes was done by examining three main aspects of conducting a classification: time, accuracy and consistency.

The time aspect was evaluated from two perspectives, learning time and classification time. Classification time refers to the time required to classify a defect. This time should ideally be kept low, as classification of a defect may be seen as an additional activity that is not strictly necessary for resolving the defect. Furthermore, as defects may be reported by a variety of project stakeholders, it is important that a classification scheme can be learnt quickly. As each defect report was designed to be comparable in scope, a decrease in classification time for each defect will be taken as evidence of the learning time.

The classification accuracy aspect was also assessed from two perspectives. Firstly, the amount of correctly classified attributes was compared between the two schemes. The comparison was made in order to evaluate whether a DCS adapted to the target domain facilitated a more accurate classification than generic DCS. Secondly, the accuracy of each experimentation group was compared to randomly generated classifications. This comparison was done to assess how reliable the classification data was.

Finally, the consistency of classification within each experiment group was examined. Consistency is important as a large variation in how subjects classify the defects would indicate that the classification is dependent on who performs the classification rather than on properties of the defects.

In the following subsections, we report on the experiment sample and population, and instrumentation. The instrumentation includes the experiment material, dependent and independent variables, and, hypotheses and analysis methods. Finally, we elaborate on the validity of study.

Subjects and Population

The subjects in this experiment were students (convenience sampling). The experiment was conducted in two sessions, in which 50 subjects participated. In the first session, 19 first year master students (i.e. in their 4th year of university studies) attending Software Engineering and Management programme participated. In the second session 31 first year bachelor students from the same programme participated.

The population of this experiment is software engineers working with implementation and quality assurance of automotive safety software. As the subjects generally lack the industrial experience and domain knowledge which can be expected for the population, we consider—based on our previous work (Staron, 2007)—the subjects as a worst-case sample.

Instrumentation

The same instrumentation—the experiment material and procedures—was used in both experiment sessions (Table 16 summarizes the experimentation material used[14]). The procedure followed was:

1. A shared 20 minute long lecture providing a general introduction to defect classification and to automotive software and their development was given to the subjects;
2. Each experimentation group received a document with a detailed description of the classification scheme. This document differed between the experimentation groups only in the classification scheme it described. The subjects were asked to read the whole document before beginning the tasks;
3. A shared set of five defect reports were classified by the subjects. For each defect, a pre-printed answer sheet was to be filled in. The answer sheet contained a field for the time started, fields for each attribute of the classification scheme, a classification confidence assessment (five-point Likert), and a field for noting the finish time;
4. After all five tasks were finished the subjects were asked to fill in a background assessment form, and an experiment evaluation form

The experiment was designed to take approximately 2 hours.

[14] The full set of experimentation material can be downloaded from: http://www.cse.chalmers.se/~nikmel/DCS/instruments.zip

Table 16. Summary of experiment objects

Introduction	*Lecture*	An introductory lecture was given to both experiment groups. The lecture described the principles of defect classification, and an overview of automotive active safety features and the development process
	Detailed introduction	A detailed written description of the specific classification scheme (ODC or LiDeC) was provided to the experiment groups. The description included an overview of the classification scheme, an example defect report and its classification, a detailed description of the classification scheme, a detailed description of the attributes and values of the scheme, and a selection of slides from the introductory lecture
Experiment	*Experiment tasks*	The experiment tasks consisted of five defect reports that were to be classified by the subjects. The defect reports were based on real defect reports selected from the same project as the case study was conducted on. The defect reports were abbreviated considerably and edited to maintain confidentiality
	Answer sheet	For each experiment task, a pre-printed answer sheet was filled in. The sheet contained a field for the time started, a field for each attribute of the classification scheme, a field for assessing the confidence with the classification (a five-point Likert scale), and finally a field for the finish time of the classification
Background and Experiment Evaluation	*Background assessment*	After the experiment, the subjects were to fill in a background assessment form, and an experiment evaluation form. The background assessment consisted of four questions assessing, on a five-point Likert scale the subject's experience with industrial software engineering, programming, defects and software quality, and automotive software
	Experiment evaluation	The experiment evaluation assessed whether the tasks were understood, how difficult the tasks were perceived (Likert). In addition, for each attribute in the classification scheme, the subject was to assess their classification confidence

Table 17. Summary of dependent variables measured in the experiment. The column Measure Type denotes whether the variable was measured directly in the experiment or calculated (derived) from directly measured variables

Aspect	Variable	Measure Type	Unit / Values	Description
Time	T_d	Derived	Minutes	T_d denotes the time required per subject to classify defect d. The variable was calculated from the start and end time noted by the subjects for each defect
	T_{tot}	Derived	Minutes	T_{tot} denotes the total time for classifying all five defects per subject. The variable was calculated as the sum T_1 to T_5 for each subject
Classi-fication	Cl_{ad}	Direct		Cl_{ad} denotes the classification of attribute *a* for defect *d*
Accuracy	Co_{ad}	Derived	Booelan	Co_{ad} denotes whether the classification Cl_{ad} was correct. This variable was derived by comparing each Cl_{ad} with a classification key we had created as part of the experiment design
	Co_d	Derived	ODC: {0-8} LiDeC: {0-10}	Co_d denotes the total number of correctly classified attributes for defect d
	Co_{tot}	Derived	ODC: {0-40} LiDeC: {0-50}	Co_{tot} denotes the total number of correctly classified attributes for all five defects
Confi-dence	Cf_d	Direct	Likert {1-5}	Cf_d denotes the self-assessed confidence in classifying defect d
Background	B_{SW}	Direct	Likert {1-5}	Self-assessed amount of previous industrial experience in software engineering
	B_P	Direct	Likert {1-5}	Self-assessed amount of previous experience in programming
	B_{SQ}	Direct	Likert {1-5}	Self-assessed amount of previous experience with defect management and software quality
	B_{ASW}	Direct	Likert {1-5}	Self-assessed amount of previous experience with automotive software development
Experiment evaluation	B_D	Direct	Likert {1-5}	Perceived difficulty of the experiment as a whole
	B_U	Direct	Tasks {1-5}	Self-assessed list of tasks that the subject did not understand
	Cf_a	Direct	Likert {1-5}	Self-assessed confidence with classifying attribute *a*. This assessment reflects the subject's confidence for each attribute of the classification scheme, over all five defects

A summary of the variables used in the analyses of the experiment results is presented in Table 17. Time per defect was directly measured by the subject noting the start and end time of each defect classification (a clock with current time was projected throughout the experiment sessions). The two time variables shown in Table 17 were derived from these measures. The subjects were asked to classify each defect in order, and to note start time before reading the defect text.

The classification of each attribute consisted of noting the value code (as shown in appendix A in (Mellegård et al., 2012b) for LiDeC, and in (IBM, 2012) for version 5.11 of ODC). The accuracy variable Co_{ad}, showing whether attribute a for defect d was classified correctly, was derived by comparing the classification code to the classification key developed by us in conjunction with the defect reports. The variables Co_d and Co_{tot} were derived by counting the number of correctly classified attributes per defect, and over all five defects respectively. In addition, for each defect, the subjects were to assess their overall classification confidence on a five-point Likert scale.

The background of each subject was, as can be seen in Table 17, assessed by four questions, each recorded on a five-point Likert scale. The questions aimed at assessing generic software engineering knowledge (industrial software engineering and programming experience), and knowledge specific for the experiment (defect management and software quality, and automotive software development).

Finally, the experiment evaluation consisted of a self-assessment of whether each of the five tasks (B_U in *Table 17*) were understood, how difficult the experiment as a whole was perceived (B_D in *Table 17*), and for each attribute of the classification scheme, how confident the subject was in classifying it (Cf_a in *Table 17*).

Hypotheses and Analysis Methods

The hypotheses evaluated by the experiment are summarized in Table 18 and Table 19. The hypotheses in Table 18 aim at comparing the two data sets (MSc and BSc students) to evaluate whether differences in experience and skill affect classification performance. The hypotheses in Table 19 aim at comparing the two experiment groups (ODC and LiDeC) to evaluate whether the type of DCS affects the classification performance.

As the variables could not be shown to fit a normal distribution, the non-parametric method Mann-Whitney's U-test was used to evaluate differences between the two data sets and between the two experimentation groups. All inferential statistics, except for hypothesis DCS_α_a, were generated using

IBM SPSS Statistics for Windows, Version 21.0 (SPSS, 2013). All descriptive statistics were generated using SPSS, except the bar chart for hypothesis DCS_C_{od} which was created using LibreOffice Calc, and the bubble diagram for hypothesis DCS_α_a which was generated using R (R Core Team, 2012)[15]. For hypothesis DCS_Co_{Rnd}, 50 random classifications for each classification scheme was generated using R (R Core Team, 2012)[16]. For hypothesis DCS_α_a, Krippendorff's α was calculated using bootstrapping with 1000 iterations. Krippendorff's alpha was calculated in R (R Core Team, 2012) with the package irr (Gamer et al., 2012), and the bootstrapping provided by kripp.boot (Gruszczynski, 2013).

[15] The R script for generating the bubble diagrams can be found here: http://www.cse.chalmers.se/~nikmel/DCS/generateBubbleDiagram.zip

[16] The R script for generating the random classification can be downloaded from: http://www.cse.chalmers.se/~nikmel/DCS/GenerateRandomClassifications.R

Table 18. Summary of hypotheses for examining the two data sets (BSc vs MSc students). These hypotheses are indented to provide information on the effect of experience on the classification performance

	Hypothesis	*Var.*	*Description*	*Analysis method*
Background	EXP_B_x	B_{SW}, B_P, B_{SQ}, B_{ASW}, B_D	H_0: *There is no difference in each of the background variable between the two data sets.* H_0 was tested for each of the background variables and used to characterize the differences in experience between the subjects in the two data sets	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
Confidence	EXP_Cf_a	Cf_a	H_0: *There is no difference in the perceived confidence between the two data sets for each attribute* H_0 was tested for each variable of each classification scheme. The result is used to characterize whether experience has any impact on the understandability of each attribute in the two classification schemes	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
	EXP_Cf_d	Cf_d	H_0: *There is no difference in the perceived classification confidence between the data sets for each defect* H_0 was tested for each defect regardless of which classification scheme that was used. The result is used to characterize whether experience has any impact on the understandability of each defect	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
Time	EXP_T_{tot}	T_{tot}	H_0: *There is no difference between the two data set in the total amount of time needed for the experimentation tasks* H_0 was tested to evaluate if experience had effect on the time required to classify the defects	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
Accuracy	EXP_Co_{tot}	Co_{tot}	H_0: *There is no difference in accuracy between the two data set in the total number of correctly classified attributes* H_0 was tested for each classification scheme separately, and used to evaluate if experience can be shown to have an effect on the precision of classification	Box plots are used to provide descriptive statistics

Table 19. Summary of hypotheses for examining the two treatments (ODC vs LiDeC). These hypotheses are indented to provide information on the effect of the classification scheme on the classification performance

	Hypo-thesis	Var.	Description	Analysis method
Background	DCS_B_x	B_{SW}, B_P, B_{SQ}, B_{ASW}, B_D	H_0: *There is no difference in each of the background variable between the two experiment groups* H_0 was tested for each of the background variables and used to evaluate whether the two experiment groups backgrounds were comparable. Any differences may affect the experiment validity	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
Confidence	DCS_Cf_d	Cf_d	H_0: *There is no difference between experiment groups in the perceived classification confidence per defect* The result is used to characterize whether the DCS used has any impact on how certain each subject was about the classification. Any differences may be explained by properties of the defect	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
Time	DCS_T_{tot}	T_{tot}	H_0: *There is no difference between the experiment groups in the total amount of time needed for the experimentation tasks*. H_0 was tested to evaluate if the DCS had an effect on the time required to perform the classification	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0
	DCS_T_d	T_d	By plotting the classification time per defect and experiment group the time to learn each classification scheme can be assessed	A decreasing slope is taken as evidence of a learning curve
Accuracy	DCS_Co_d	Co_d	H_0: *There is no difference in accuracy between the experiment groups in the amount of correctly classified attributes per defect* As the two classification schemes have different number of attributes, the percentage of correctly classified attributes per subject and defect was calculated. H_0 was tested to evaluate whether the DCS had an effect on the precision of classification	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0 A bar chart is used to provide descriptive statistics
	DCS_Co_{Rnd}	Co_{tot}	H_0: *There is no difference in accuracy between each of the experiment groups and a random classification using the same DCS* The hypothesis was tested in order to evaluate whether the classification accuracy was due to factors relating to properties of the defect reports. If the hypothesis cannot be rejected – meaning that a human classifier is no more accurate than chance – it would suggest that the classification data is not reliable. H_0 was tested for each classification scheme separately	Mann-Whitney's U-test $p < 0.05$ is required to reject H_0 Box plots are used to provide descriptive statistics
Consistency	DCS_α_a	Cl_{ad}	H_0: *There is no good agreement among the subjects in their classification of each attribute* The inter-rater agreement was calculated in order to assess the classification consistency. Consistency is desirable to minimize the dependency on who is performing the classification. In addition, bubble diagrams were generated for two attributes of LiDeC to illustrate the distribution of values. The choice if attributes was done based on experiences from the case study	Krippendorff's alpha statistic for inter-rater agreement $\alpha > 0.667$ is required to reject H_0 Bubble diagram for descriptive statistics

Validity Evaluation

The main threats to the validity fall into the categories internal, external, construct and conclusion validity (Wohlin et al., 2000). We now briefly discuss the main threats to each of these categories of validity.

Internal validity – Internal validity concerns unforeseen influences on the independent variables. Such influences may pose a threat to the validity of conclusions about the causal relationship between treatment and outcome. The main threat to internal validity, in this experiment, would be inappropriately designed instrumentation. For example, it was considered important (for external validity) to use defect reports that resembled real reports as closely as possible. This, however, might risk making them too domain specific to be understandable to non-domain experts. In order to minimize the threat to internal validity a pilot experiment with seven PhD students was conducted. The pilot experiment included eight defect reports. The instrumentation was evaluated after the experiment, which resulted in a number of modifications to the material—e.g. in the data collection forms—and selecting the five defect reports that were considered most readily understandable;

External validity – External validity relates to whether the results and conclusions can be generalized to a context outside of the experiment, which in our case is industrial automotive software design. The main threats in our case relate to selection and setting (Wohlin et al., 2000), i.e. the selection of subjects and the experimentation material used. The use of student subjects poses a threat to the generalizability of the results, as the subjects might not be representative to the population. However, based on previous research student subject can be seen as a worst-case sample. In an industrial setting, it can be assumed that the population has considerably more domain knowledge (which we found indications on being a factor in both accuracy and consistency). The experimentation material (the defect reports, ODC and LiDeC) were selected and adapted from actual industrial cases, making the experimentation material representative for the target context;

Construct validity – Construct validity refers to whether the experiment instrumentation is appropriate for the theory it was designed to measure. In our case, the main threats to construct validity concerns mono-method bias and confounding constructs and level of constructs (Wohlin et al., 2000).

- *Mono-method bias* – In the experiment we measured several aspects of classification performance (time, accuracy and consistency). Furthermore, results from the experiment and case study can be

crosschecked to validate the results. A possible source of mono-method bias is the classification key that was developed by the researchers and used to determine whether a classification was correct. As the classification key was developed by the researchers in conjunction with the defect reports, it is assumed to match appropriately. The key, however, was not validated with practitioners, thus there might be a risk of introducing researchers bias in the accuracy variables;

- *Confounding constructs and levels of constructs* – There is a risk that the extent of the subjects' prior knowledge (e.g. programming skills) may influence their performance. To investigate this, the experiment was conducted in two sessions with groups of students; one group with more experience subjects. The results of these two groups were compared, and differences elaborated on.

Conclusion validity – Conclusion validity concerns threats to the ability to draw the correct conclusions about the relationship between treatment and outcome. As the measured variables were not normally distributed, the less powerful non-parametric tests were used. This may increase that risk of making type-II errors. We minimize this threat to validity by being careful to draw conclusions based on inability to reject the null hypothesis. An additional threat to conclusion validity is random heterogeneity of subjects. However, as all subjects were university undergraduate students, they can be assumed to have similar knowledge and background. In order to validate this assumption a background questionnaire was submitted by each subject.

8.4.2 Part II: Case study

The industrial evaluation of LiDeC was originally the final part of a three-stage case study (Mellegård et al., 2012a). Whereas the two initial stages aimed at adapting IEEE Std. 1044 to the development context of the case company (resulting in LiDeC), the final stage aimed at validating the adapted DCS. The validation consisted of classifying a sample of defects from a finished project.

The case study method was considered appropriate as the applicability of the adapted classification scheme was of importance. The methodology of the study is based on Yin (2009), specifically, a single-case design by applying the classification scheme to defect reports from one project at our case company.

In the case study, four subjects involved in the project participated in six separate two-hour long sessions (two subjects participated in two consecutive

classification sessions) in which 22 defects were classified. All defects had been reported in the project, which had finished one year prior to the study and contained several hundred defect reports[17]. All subjects involved in the study had been part of the project with the following roles: two developers, one tester and one project leader.

The 22 defects were selected randomly using blocking. The defect reports were divided into two blocks, where the peak close to software release (as shown in Figure 28) was assigned as one block and the remainder as the second block. In the analyses reported in this paper, only defects from the peak were used. This decision was made because of an expressed interest by the company to examine those defects more closely. The sample constituted approximately 10% of the defects from the peak.

The results from the case study included classification data from the selected sample of defects. The collected data was analysed and the results presented at a company workshop for validation. Additionally, the time required to classify each defect was recorded. The classification time was plotted in order to provide an initial estimate on learning and classification time requirements. This chart is, in this paper, compared to the results from the experiment—any similarities will be used to argue in favour for the external validity of the experiment results.

8.5 RESULTS

The results from the experiment and the case study are presented in the following two subsections.

8.5.1 Experiment Results

In this section, we present an analysis of the data from two perspectives. Firstly, the experiment was conducted in two sessions—forming two data sets—one with master students (MSc) as subjects and one with bachelor students (BSc). This presents the opportunity to examine whether the amount of acquired knowledge and skill affect classification performance (the related hypotheses are presented in Table 18). In addition, if significant differences were found, it might not be appropriate to conduct analyses on the merge of the two sets. Secondly, aspects pertaining to the type of classification scheme used were analysed. Specifically, classification learning time, accuracy and consistency of classification were examined for the two schemes (the related

[17] Exact number cannot be disclosed due to confidentiality reasons

hypotheses are presented in Table 19). Each of these two perspectives are presented in the two subsequent sections.

From analysing the B_U variable, one subject (in the BSc/LiDeC group) stated that they did not understand any of the tasks. The subject, furthermore, classified only one defect and did not submit the evaluation form. Therefore, the subject was removed from the data.

Missing data was excluded pair-wise in the hypothesis analyses.

Impact of Experience

Statistically significant differences between the two data sets were found. As can be seen in Table 20, showing the comparison of the background assessments from the two data sets, there was a significant difference between the data sets in terms the subjects' self-assessed prior experience. The differences, which were limited to generic software engineering knowledge, were anticipated as the MSc students, in addition to being further advanced in their studies, have to a larger extent industrial experience. This difference may have effect on the subjects' ability to understand the experiment material. There was, however, no significant difference in the subjects' experience with automotive software systems, which might have had greater effect on the understanding of the experiment material. There was also no significant difference in the perceived difficulty of the experiment material. Thus indicating that the differences in generic software engineering experience may have had limited impact on how difficult the material was to assimilate.

Table 20. Results of testing hypothesis EXP_B_x, i.e. the differences in background assessment and perceived difficulty between the two data sets (MSc vs BSc students). Statistically significant results on the 0.05 level have been highlighted

	Mann-Whitney U	*Asymp. Sig. (2-tailed)*	*Reject H_0*
Industrial Exp (B_{SW})	213.500	**0.050**	**Yes**
Programming Exp (B_P)	83.000	**0.000**	**Yes**
Defects and SQ Exp (B_{SQ})	135.500	**0.000**	**Yes**
Automotive SW Exp. (B_{ASW})	278.000	0.558	No
Difficulty (B_D)	258.500	0.423	No

A comparison of the subjects' perceived confidence per attribute is shown in Table 21 and Table 22 (for ODC and LiDeC respectively). As can be seen in the tables, there were no differences in perceived confidence for the attributes in LiDeC, while there were two attributes in ODC that showed significant difference.

Table 21. Results of hypothesis EXP_Cf_a (for the ODC group), comparing perceived confidence per attribute between the MSc and BSc student groups (statistically significant results on the 0.05 level have been highlighted)

ODC	*Mann-Whitney U*	*Exact Sig. [2*(1-tailed Sig.)]b*
Activity	56.500	0.220
Trigger	54.500	0.182
Impact	51.000	0.135
Target	52.000	0.150
Source	41.000	0.041
Age	58.000	0.262
Type	37.000	0.023
Qualifier	45.500	0.068

[b] Not corrected for ties

Table 22. Results of hypothesis EXP_Cf$_a$ (for the LiDeC group), comparing perceived confidence per attribute between the MSc and BSc student groups

LiDeC	*Mann-Whitney U*	*Exact Sig. [2*(1-tailed Sig.)]*[b]
Detection	42.500	0.201
Urgency	51.500	0.477
Severity	40.500	0.159
Effect	62.500	0.975
Artefact	44.500	0.250
Injection	49.000	0.403
Type	58.500	0.781
Product Impact	52.500	0.516
Verification level	39.500	0.141
Resolution State	37.500	0.109

[b] Not corrected for ties

A comparison between the MSc and BSc students in terms of their perceived classification confidence per defect is shown in Table 23. As can be seen, the MSc students were significantly more confident than the BSc students in their classification of defect 2.

Table 23. Results for hypothesis EXP_Cf_d, comparing perceived classification confidence per defect, MSc vs BSc students (statistically significant results on 0.05 level have been highlighted)

	Data set	*N*	*Mean Rank*	*Mann-Whitney U*	*Asymp. Sig. (2-tailed)*
Defect1	MSc	19	29.03		
	BSc	31	23.34		
	Total	50		227.500	0.156
Defect2	MSc	19	30.71		
	BSc	31	22.31		
	Total	50		195.500	**0.035**
Defect3	MSc	19	26.74		
	BSc	30	23.90		
	Total	49		252.000	0.477
Defect4	MSc	19	28.74		
	BSc	31	23.52		
	Total	50		233.000	0.197
Defect5	MSc	19	27.74		
	BSc	31	24.13		
	Total	50		252.000	0.376

In terms of the time required to classify defects, there was no significant difference. Table 24 shows the comparison of the total time required to classify all five defects.

Table 24. Results for hypothesis EXP_T_{tot}, analysis of total time required for classifying all 5 defects, MSc vs BSc students

	TotalTime
Mann-Whitney U	288.500
Asymp. Sig. (2-tailed)	0.904

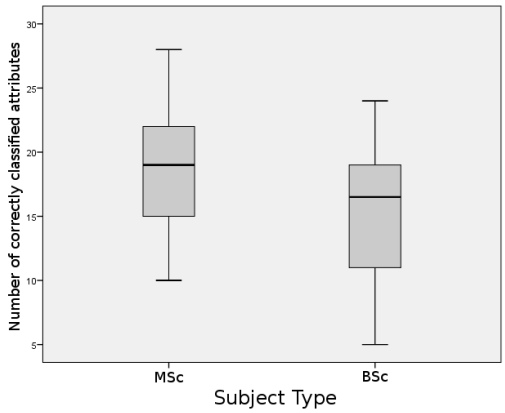

Figure 20. Results for hypothesis EXP_Co$_{tot}$, comparing accuracy between the MSc and BSc datasets (for ODC)

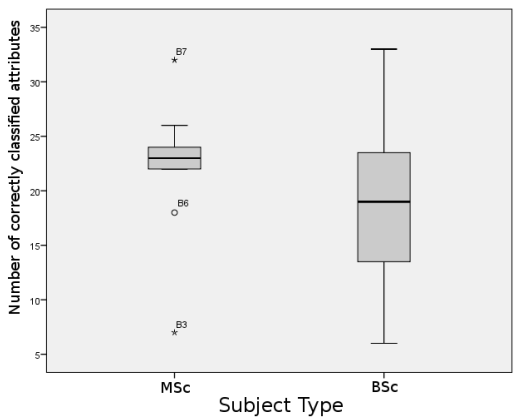

Figure 21. Results for hypothesis EXP_Co$_{tot}$, comparing accuracy between the MSc and BSc datasets (for LiDeC)

A comparison of the classification accuracy between the BSc and MSc students is shown in Figure 20 and Figure 21 for ODC and LiDeC respectively. In both figures, the vertical axis denote the total number of correctly classified attributes over all five defects[18]. As can be seen in the figures, the mean value seem to be slightly lower for the BSc students for

[18] Please note that, as the number of attributes in ODC and LiDeC differs, the absolute number (as shown in the vertical axis in each figure) is not comparable between the schemes

both schemes. The variance within the samples, however, is large (with the exception of MSc students using LiDeC, shown to the left in Figure 21), thus the apparent difference in means is not statistically significant.

The notably lower variance within the MSc students classifying according to the LiDeC scheme might indicate that more experienced subjects in combination with a classification scheme adapted to the specific domain of the defect reports contribute to a more uniform classification performance. Extrapolating this to an industrial context, it would suggest that an adapted classification scheme increases the consistency of the classification and therefore also the reliability of the classification data.

In summary, as the differences found between the datasets were in the students own assessments (background and confidence) while no differences could be shown for the main dependent variables (time and accuracy) the two datasets were merged. All subsequent analyses were done on the merged dataset.

Impact of Classification Scheme

In order to evaluate the threat to internal and conclusion validity, the background survey for each experimentation group was analysed. If significant differences between the groups were found it would confound any effects found in the experiment, as they may not necessarily be attributable to the treatments provided to the experimentation groups.

Table 25. Results of testing hypothesis DCS_B_x, i.e. the differences in background assessment and perceived difficulty between the experiment groups (ODC vs LiDeC)

	Mann-Whitney U	*Asymp. Sig. (2-tailed)*
Industrial Exp (B_{SW})	269.000	0.312
Programming Exp (B_P)	284.000	0.558
Defects and SQ Exp (B_{SO})	274.000	0.393
Automotive SW Exp (B_{ASW})	292.000	0.491
Difficulty (B_D)	233.000	0.088

As can be seen in Table 25, no statistically significant differences in the subjects' background survey could be shown. However, the difference between the groups in their perceived difficulty of the experiment (shown in

Table 25 as *Difficulty*) was close to being significant at the 0.05 level, where the ODC group considered the experiment more difficult than the LiDeC group. This difference can be attributed the fact that LiDeC is adapted to the same technical domain as the defect reports, and might therefore be perceived less difficult to apply than the more generic ODC.

Table 26. Results for hypothesis DCS_Cf$_d$, comparing perceived classification confidence per defect, ODC vs LiDeC (statistically significant results on 0.05 level have been highlighted)

	Group	*N*	*Mean Rank*	*Sum of Ranks*	*Mann-Whitney U*	*Asymp. Sig. (2-tailed)*
Defect1	ODC	26	24.40	634.50		
	LiDeC	24	26.69	640.50		
	Total	50			283.500	0.558
Defect2	ODC	26	22.27	579.00		
	LiDeC	24	29.00	696.00		
	Total	50			228.000	0.082
Defect3	ODC	25	24.88	622.00		
	LiDeC	24	25.13	603.00		
	Total	49			297.000	0.950
Defect4	ODC	26	21.17	550.50		
	LiDeC	24	30.19	724.50		
	Total	50			199.500	**0.022**
Defect5	ODC	26	23.42	609.00		
	LiDeC	24	27.75	666.00		
	Total	50			258.000	0.274

As can be seen in Table 26—showing the subjects' self-assessed confidence per classified defect—the LiDeC group was significantly more confident in their classification of defect 4.

In summary, no significant differences in the subjects' self-assessed background experiences were found that would pose a threat to the internal validity of the experiment. Therefore, it will be assumed that the effects seen

in the subsequent analyses can be attributed to the differences in the classification schemes used (the treatment).

Classification time

As can be seen in Table 27, there was no significant difference between the groups in the total amount of time required to classify all five defects ($p>0.05$). Presumably, the time required to read and understand each defect report outweighs any differences that may have been contributed by the classification schemes themselves.

Table 27. Results for hypothesis DCS_T_{tot}, analysis of total time required for classifying all 5 defects, ODC vs LiDeC

	TotalTime
Mann-Whitney U	258.000
Asymp. Sig. (2-tailed)	0.294

As can be seen in Figure 22—showing classification time per defect—both the average time required per defect and the variance within each group is similar. This corroborates the conclusion that there is no significant difference between the groups. What can be seen in Figure 22, however, is a clear indication of a learning curve, where the initial defects require considerably longer time than the subsequent ones.

The classification time, as seen in Figure 22, seems to converge at approximately 4 to 5 minutes per defect. In the experiment, this includes the time required for the subjects to read the defect report, suggesting that the actual classification time may be only a fraction of that. In an industrial setting, a defect would be classified at the time of reporting when the reporter would already have detailed knowledge about the defect. Thus, the additional time required to classify a defect will most likely be small.

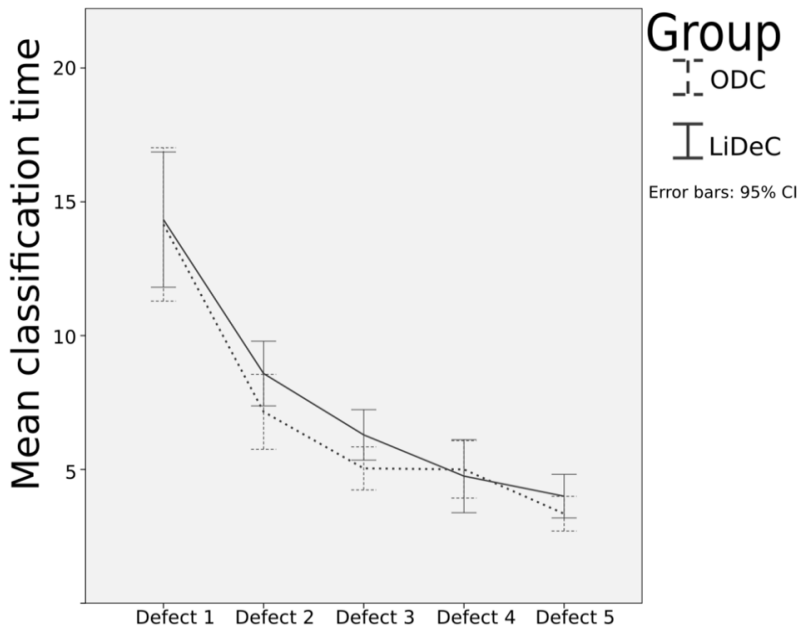

Figure 22. Results for hypothesis DCS_T_d, showing classification time per defect for each of the two classification schemes. A decrease in classificaion time indicate learning time

Accuracy

The accuracy of the classification was examined in two ways. Firstly, the amount of correctly classified attributes was compared between the schemes. Secondly, the accuracy of each scheme was compared to random classifications.

Figure 23 shows the classification accuracy (as percentage of correctly classified attributes) for each defect, and Table 28 shows the statistical analysis of the difference between the groups (using Mann-Whitney's U-test). As can be seen in Figure 23, the classification accuracy (varying between 31% and 53%) appears similar for both schemes. One interesting observation is that the accuracy of the LiDeC groups appears higher for defect 4, which is corroborated by the analyses shown in Table 28—only defect 4 showed a statistically significant difference. Defect 4 also had (as can be seen in Table 26) a significantly higher perceived classification confidence; thus suggesting that the higher perceived confidence is justified.

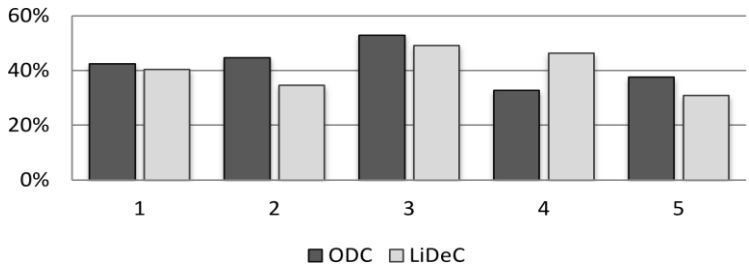

Figure 23. Bar chart for hypothesis DCS_Co$_d$, representing the amount of correctly classified attributes per defect (ODC vs LiDeC)

Table 28. Results for hypothesis DCS_Co$_d$, comparing the accuracy as the percentage of correctly classified attributes per defect (ODC vs LiDeC)

	Group	*N*	*Mean Rank*	*Sum of Ranks*	*Mann-Whitney U*	*Asymp. Sig. (2-tailed)*
Defect1	ODC	26	26.04	677.00		
	LiDeC	24	24.92	598.00		
	Total	50			298.000	0.782
Defect2	ODC	26	28.90	751.50		
	LiDeC	24	21.81	523.50		
	Total	50			223.500	0.083
Defect3	ODC	26	26.60	691.50		
	LiDeC	24	24.31	583.50		
	Total	50			283.500	0.577
Defect4	ODC	26	20.44	531.50		
	LiDeC	24	30.98	743.50		
	Total	50			180.500	**0.010**
Defect5	ODC	26	27.65	719.00		
	LiDeC	24	23.17	556.00		
	Total	50			256.000	0.275
Total	ODC	26	25.87	672.50		
	LiDeC	24	25.10	602.50		
	Total	50			302.500	0.853

Figure 24 and Figure 25 show the accuracy of a human classifier compared to classifications done at random. As can be seen in the figures, human classifiers performed significantly better than a random classification (p<0.01 using Mann-Whitney U).

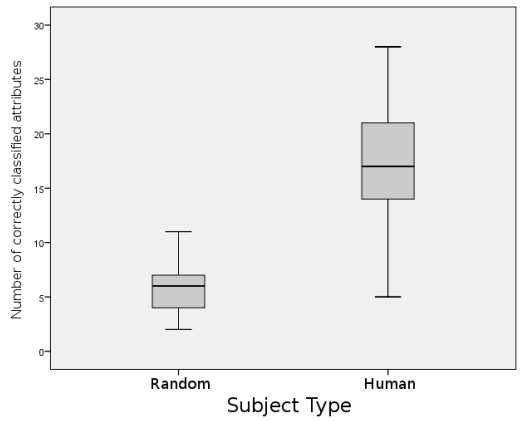

Figure 24. Results for hypothesis DCS_Co$_{Rnd}$ (ODC), comparing the accuracy of a human classification to that of a randomly generated one (ODC). The difference is statistically significant (p<0.01)

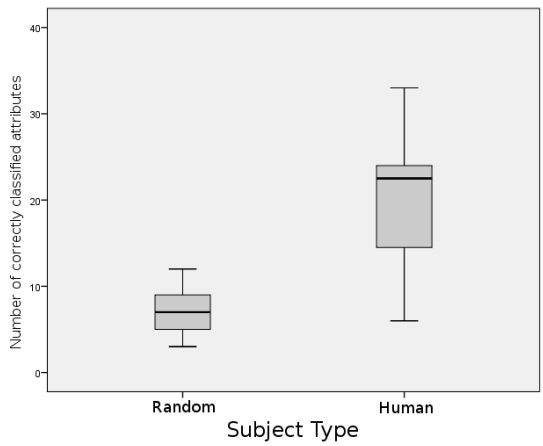

Figure 25. Results for hypothesis DCS_Co$_{Rnd}$ (LiDeC), comparing the accuracy of a human classification to that of a randomly generated one. The difference is statistically significant (p<0.01)

Consistency

The classification consistency was examined using the Krippendorff's alpha statistic. As can be seen in Table 29 and Table 30, showing the alpha statistic for each attribute in ODC and LiDeC respectively, the calculated inter-rater agreement is low. Krippendorff (2004, p. 241) recommends alpha values to

exceed 0.667 in order to be considered as acceptable, and to exceed 0.8 to be considered a (near) perfect match. This low agreement statistic is in line with one previous study (Henningsson and Wohlin, 2004), but also in contrary to the findings of others (El Emam and Wieczorek, 1998; Freimut et al., 2005). We elaborate further on this in section 8.6.2.

Table 29. Results for hypothesis DCS_α_a, showing the Krippendorff's alpha statistic for the ODC attributes with 5 defects and 26 subjects

Attribute	*Alpha*
Activity	0.241
Trigger	0.103
Impact	0.008
Target	0.128
Source	0.042
Age	0.028
Type	0.201
Qualifier	0.124

Table 30. Results for hypothesis DCS_α_a, showing the Krippendorff's alpha statistic for the LiDeC attributes with 5 defects and 24 subjects

Attribute	*Alpha*
Detection	0.268
Urgency	0.191
Severity	0.087
Effect	0.108
Artefact	0.070
Injection	0.019
Type	0.209
Product	0.051
Verification	0.083
Resolution	0.082

As can be seen in Table 30, the detection attribute has the highest alpha, while the injection attribute has the lowest. Figure 26 and Figure 27 illustrate the distribution of answers for these two attributes respectively. In the figures, the horizontal axis represent each of the five defects in the experiment, the vertical axis represent the available values for the attribute, and the area of each bubble is proportionate to the number of subjects that assigned the value to the defect.

In Figure 26, it can be seen that for each defect (with the exception of defect 4) one bubble is considerably larger than the others are, while in Figure 27 the bubbles are more similar. This indicates indeed a higher degree of agreement among the subjects for the *Detection* than the *Injection* attribute. This is, futhermore, in line with a finding from the industrial case study, were the *Injection* attribute was removed. The reason was that the subjects considered it too difficult to deduce the necessary information from the defect reports to reliably assign the *Injection* attribute.

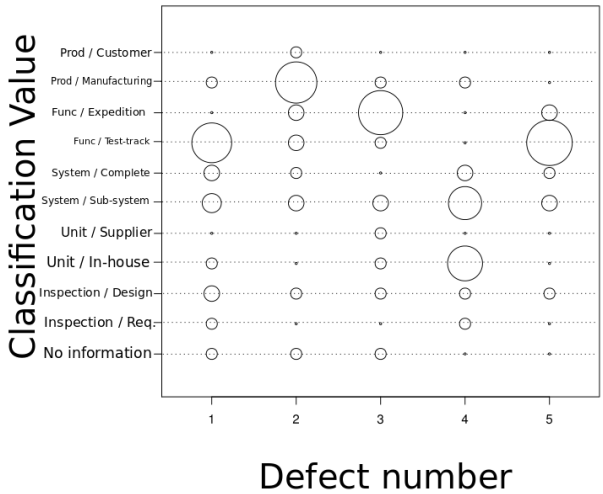

Figure 26. Bubble diagram for hypothesis DCS_α_a, illustrating the distribution of assigned values per defect for the Detection activity attribute in LiDeC. The size of each bubble is proportional to the number of subjects that assigned that value for each defect.

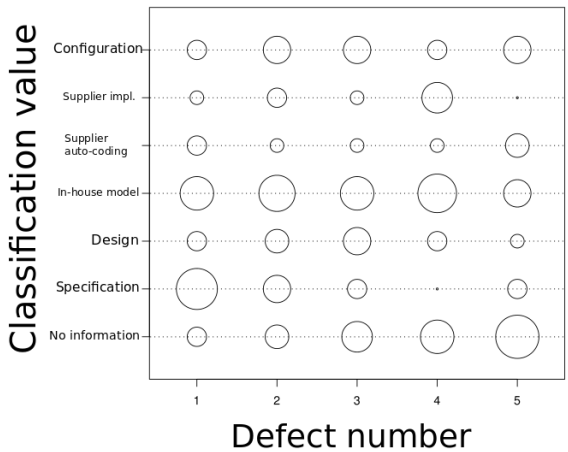

Figure 27. Bubble diagram for hypothesis DCS_α_a, illustrating the distribution of assigned values per defect for the Injection activity attribute in LiDeC. The size of each bubble is proportional to the number of subjects that assigned that value for each defect.

8.5.2 Case Study Results

In this section, we report on the results from the industrial case study. The aim of the study was to characterize defects found late in projects. Specifically to provide evidence of whether the defects were caused by integration issues, as was anticipated by expert opinion. Additionally, we examined the time required to perform the classification in order to evaluate classification efficiency.

Cause of Late Defects

To complement the broad experiment, we conducted an in-depth case study (Mellegård et al., 2012a) of in-process defects from one system developed at the department. We found, in the initial analyses of the defect data, that there was a substantial inflow of defects in late project phases. Figure 28 shows the number of unresolved defects through the project (the defect backlog). According to established software reliability growth models (for instance, s-shaped models such as the Rayleigh model (Kan, 1995)), a defect backlog should typically resemble a bell shaped curve. Initially, when focus is on feature development, the number of reported defects is low. In the middle of the project, focus is shifted towards shoring up the system and testing therefore intensifies, typically resulting in a substantially increased defect inflow. As testing continues and defects are resolved, there should ideally be fewer defects in the system to find, and therefore the inflow of defects should decrease.

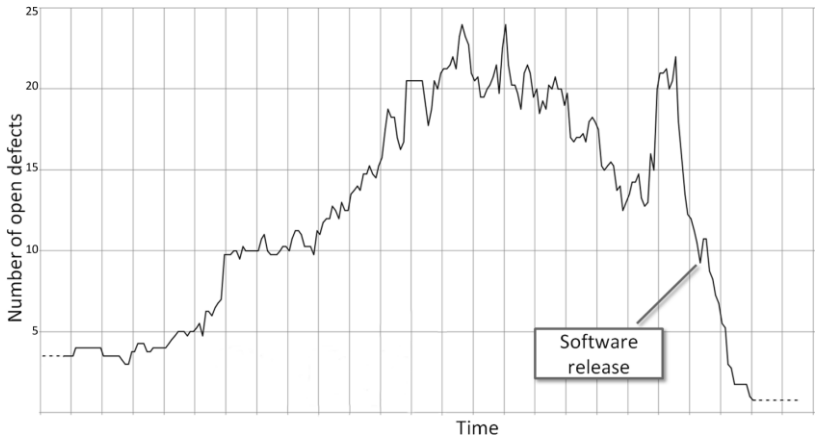

Figure 28 Defect backlog, showing the number of open defects throughout the project. Note that the total number of defects (y-axis) has been scaled to 100 and the time scale (x-axis) has been removed due to confidentiality reasons. In addition, the time scale has been cropped (indicated by the ellipsis in the start and end of the curve) and does therefore not include the last phase leading up to start of production

As can be seen in Figure 28, the shape of the backlog curve resembles the typical bell curve with one striking exception: the sharp spike in unresolved defects very late in the project. Similar spikes were found in one previous project and one that was currently on going, suggesting a systemic cause. Furthermore, the timing of the late spike, close to software release (a major in-development milestone), seemed to confirm the hypothesis of integration issues. To evaluate the hypothesis, LiDeC was applied to a sample of defects from the defect inflow spike shown in Figure 28, and the resulting classification data was analysed.

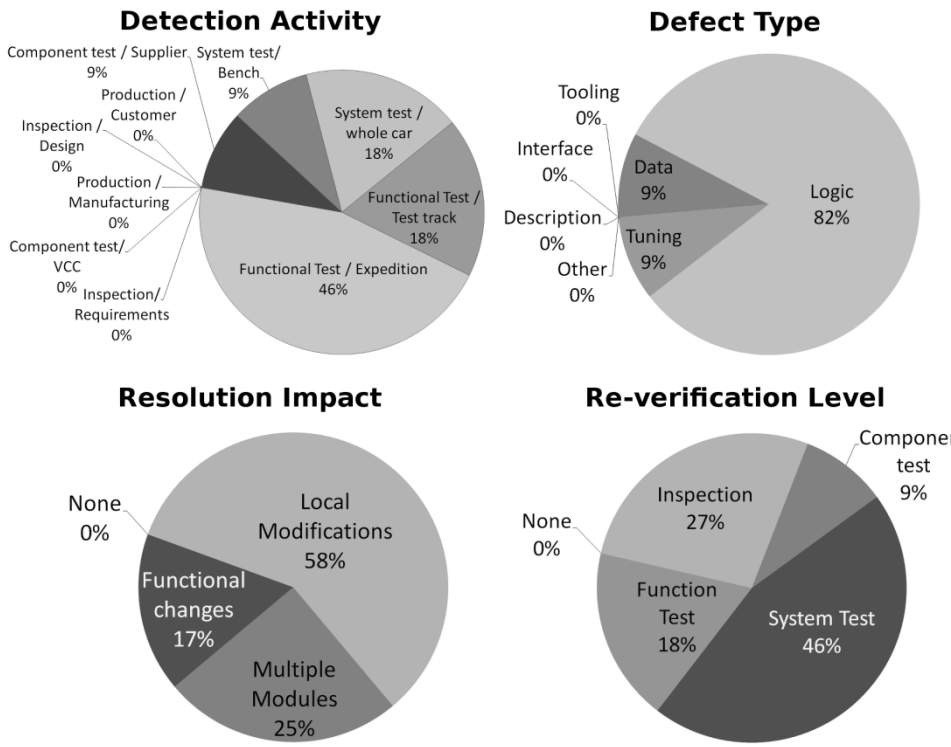

Figure 29 Preliminary analysis results, showing the distribution of classification values for the attributes Detection activity, Defect type, Resolution impact and Re-verification level

Figure 29 shows a sample of the analysis using four LiDeC attributes (see (Mellegård et al., 2012b)), providing information from three perspectives: detection of the failure, the type of the underlying fault, and finally product and project impact of applying the resolution. Using these three perspectives, the defects were examined to evaluate whether they were integration issues as anticipated.

As can be seen in the top two charts in Figure 29, while the majority of defects were indeed detected during integration testing—system or functional—the defect types were not typical for integration issues. Whereas the anticipated type would be *Interface*, *Data* or *Tuning*, the majority of defects were of the type *Logic*—i.e. computational or algorithmic faults. Such defects would normally be present already in the simulation models. This was corroborated by the *Resolution impact* attribute, shown in the

bottom left of Figure 29, indicating that most defects required changes to a single unit – integration issues would typically have an impact on multiple units, or require changes to the specification (denoted as *Functional changes*).

However, the *Re-verification Level* attribute—showing the activity required to test a resolution—indicates that it is not a clear-cut case. On the one hand, a significant amount of defects required only inspection or component test, indicating unit problems. On the other hand, most defects would require new system or functional tests, indeed indicating integration impact—finding the root cause of these defects could bring significant benefits.

Our (careful) conclusions from this study is that the majority of late defects—although to a large extent requiring new integration tests—are not of the type typically associated with integration problems. Thus, the classification of defects provided the development teams with new information that may contribute to better test planning—e.g. put effort into improving testability of requirements on unit level.

Finally, we would like to emphasize that the analysis presented here was conducted on a sample of defects from a project that had finished a year prior to the study. The results should therefore be treated as proof-of-concept rather than as a basis for recommended change of practice. We can however conclude that the classification contributed with new information that in part contradicted expert opinion. This raised interest and inspired discussions regarding possible causes; the case study provides evidence that conducting defect classification and analysis contributes to constructive review of the state-of-practice.

Classification time

In addition to the classification data, the classification time per defect was measured. In the study, two subjects participated in two classification sessions conducted more than a week apart. Figure 30, shows the classification time per defect for both subjects and session, provides an interesting comparison to the results from the experiment (see Figure 22). As can be seen in Figure 30, the initial defects in the first session by the two subjects (denoted S1.1 and S2.1 in the figure) took considerably longer than the subsequent defects; on average twice as long as in the experiment. This can be explained by the difference in size of the defects in the case study and the ones used in the experiment. As can also be seen in Figure 30, the time required to classify defects in the second session (denoted S1.2 and S2.2) was

considerably less than in the initial session, suggesting that the learning that took place in the first session was retained by the subjects.

Interestingly, in both the experiment and the case study the classification time seemed to stabilize around 5 minutes per defect. It should be noted that reading the defect description is included in that time.

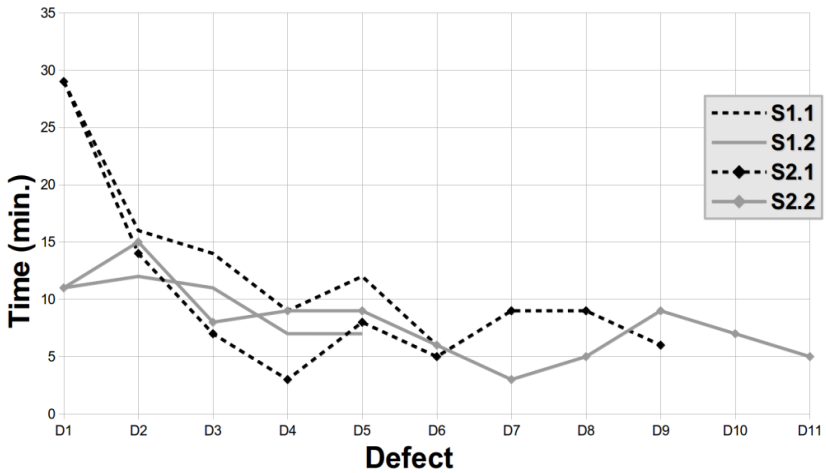

Figure 30. Classification time per defect in the industrial case study. Sx.y denote session y for subject x. Note especially the learning time for session 1 (dashed lines), and that the knowledge seems to have been retained by both subjects for session 2 (solid lines)

8.6 DISCUSSION

In this section, we discuss general observations on the impact of experience and the choice of DCS on defect classification performance. We also provide methodology reflections.

8.6.1 Observations about Defect Classification

DCSs are quickly learnt, and knowledge is retained. Although the results showed no statistically significant difference in classification time between the two DCSs, the results did show a clear indication of learning time (see Figure 22). The same pattern could also be found in the industrial case study (see Figure 30), where the classification time per defect was similar, although slightly longer. A likely reason for this is that the defect reports classified in

the case study were significantly longer than the edited ones used in the experiment. Nevertheless, results from the two studies indicate that the classification time stabilizes already after 3 – 5 defects, suggesting that the subjects are quickly able to learn the DCS. In addition, the knowledge acquired after practicing is retained. As shown by the solid lines in Figure 30, the two subjects that participated in a second classification session classified the initial defect considerably more quickly than in the first session (shown with dashes lines in Figure 30), even though there was more than a week between session.

DCS applied accurately, even without domain knowledge. In terms of accuracy, human classifications were significantly more accurate than random classifications for both DCSs (see Figure 24 and Figure 25). This shows that, even though lacking domain-specific knowledge, the subjects could still analyse the defect description and arrive at a rational classification. However, by examining the lower whisker of the right-most boxplots in Figure 24 and Figure 25, there exist subjects in both groups that do not perform better than random. This suggests, although rather trivially, that in order to ensure reliable defect classification data, experience (and perhaps training) is necessary.

Experience and domain-specific DCS contributes to classification performance. Interestingly, the variance in classification accuracy differed considerably with experience for LiDeC but not for ODC (as can be seen in Figure 20 and Figure 21). This means that the more experienced subjects using domain specific DCS perform a more consistent classification. This indicates that both the experience of the staff and the design of the DCS are factors influencing the reliability of the data.

Classification should be done in-process, and contain regular reviews. Still, the accuracy in the experiment can be considered low—approximately 40 – 50% of the attributes in both schemes were classified correctly. However, in the industrial case study, which had a similar setup (in that defect reports from a finished project were analysed post-factum), a number of attributes were considered difficult to classify because the defect descriptions were lacking the necessary information; for instance, the *Injection Activity* attribute. In these cases, the practitioners stated that the necessary information would have been known to the staff at the time of reporting. Moreover, as the classifications were analysed on a nominal scale, there is no notion of degree of correctness, even though such can be identified in practice. For instance, for the *Injection Activity* attribute in LiDeC, there may

be multiple valid classifications depending on what is considered the actual root cause. In addition to contributing with an apparent lower accuracy in the experiment, it highlights a need to establish and maintain organizational classification praxis—for example, by regularly reviewing sample classifications.

Domain-specific DCS provides more relevant guidance. As can be seen in Table 20, the MSc students had more software engineering experience than the BSc students did, while no difference in their experience with automotive software (i.e. domain knowledge) were found. Even so, there was no significant difference in perceived difficulty of the experiment, suggesting that general software engineering experience may not be a significant factor in understanding the material. Instead, it may be domain specific knowledge that is needed. As can be seen in Table 21 and Table 22, the less experienced subjects were significantly less confident in classifying two ODC attributes (*Type* and *Source*). The *Type* attribute has a quite technical (source code related) description, which might require more software engineering experience to understand. The *Source* attribute, captures data on whether the defect was found in an artefact developed in-house of outsourced. One explanation to the significant difference in confidence is that the defect descriptions use the word "supplier" to indicate that an artefact was outsourced. This may have caused the BSc students to feel less confident than the MSc students. In LiDeC, however, no such difference could be shown. This suggest that LiDeC, designed for the target development context, provides more relevant guidance; thus, indicating a benefit of adapting the DCS to the development context.

A domain-specific DCS provides more data with no extra effort. There was no significant difference in the time required to classify the defects (see Table 27), even though the LiDeC group classified two additional attributes, producing 20% more data compared to the ODC group. This indicates that the time it takes to perform the classification is small in comparison to reading the defect report (which in an industrial setting would correspond to documenting the defect). Thus, as the defect data proved useful in the case study, classifying defects can be considered a cost-effective approach to structured defect data collection.

A domain-specific DCS may increase likelihood of adoption. While no statistically significant difference in accuracy could be shown, the LiDeC group perceived the experiment as less difficult (see Table 25, where the B_D variable was close to being significant), and classified the defects that were

more domain specific with higher confidence (see defect 2 and 4 in Table 26). Thus, by adapting a classification scheme to the target domain, its perceived difficulty is reduced and the confidence of the classification is increased—at least for the defects that are more domain-specific. This, in turn, may contribute to making practitioners more inclined in adopting defect classification in the process, and retaining that practice—which is needed in order to establish and maintain company baselines.

Risk with a too specific DCS. The results described above show that there is value in adapting a DCS to the specific development context in which it is to be deployed. A concern, however, is that a too specific classification scheme risk making the collected data equally specific, and therefore risk impeding comparisons between organizations or even departments within an organization. One solution, may be to design the attribute values in a hierarchical manner, as the adapted *Impact* attribute presented by Vetro et al. (2012) and the *Detection activity* attribute in LiDeC (Mellegård et al., 2012b). In LiDeC, the top-level values in the *Detection activity* attribute are chosen such that they should be applicable to most organizations (the values include *Inspection*, *Unit*, *System*, *Function*, and *Customer* report). Each top-level value is then broken down into the specific types of detection activities that apply to the domain; the domain-specific values are used in the classification, while the top-level values are intended to provide a way to generalize the data.

Low classification consistency might indicate methodological problems. The classification consistency was found to be low for both schemes (see Table 29 and Table 30), where the highest alpha-values where between 0.201 and 0.268, far below the 0.667 threshold that Krippendorff recommends for good agreement. This apparently poor classification agreement among the subjects would suggest the classification data to be unreliable—i.e. that the classification outcome may be too dependent on who performs the classification, rather than on properties of the defects. However, the bubble diagrams generated for the LiDeC attributes with highest and lowest alpha (see Figure 26 and Figure 27) do indicate a considerable difference; whereas in Figure 27 the bubbles have similar sizes, showing no consensus in classification, in Figure 26 a clear pattern can be seen. Even though the visualization does not provide conclusive evidence, it suggests that there might be problems with the statistical method used to assess the agreement (further elaborated on in section 8.6.2).

8.6.2 Methodological Reflections

In part, we have in this paper aimed to describe a rigorous method for evaluating the efficiency and effectiveness of DCSs. In the method, the efficiency of the scheme is evaluated through a controlled experiment, while the effectiveness is evaluated using an industrial case study.

The aspects related to efficiency of a DCS were classification and learning time, accuracy and consistency—providing evidence regarding the cost of classifying defects, and the reliability of that data. While the method to analyse the first two aspects are straightforward (as described in section 8.4), the consistency aspect warrants further elaboration. Whereas previous studies evaluating the consistency of classification have used Cohen's kappa statistic, in our experiment Krippendorff's alpha was used instead; reasons include that Krippendorff's alpha can be applied for more than two subjects[19], and is more flexible in handling data with missing values (Hallgren, 2012; Hayes and Krippendorff, 2007).

Previous studies on classification consistency have delivered contradictory results, whereas the studies by El-Emam and Wieczorek (1998) and Freimut et al. (Freimut et al., 2005) showed fair to good agreement for the type attribute in ODC, the study by Henningsson and Wohlin (2004) could not show such agreement for the same attribute. It should be noted, however, that in the latter study, university students were used as subjects, whereas in the former two practitioners were used. Nevertheless, the criticism of Cohen's kappa as a coefficient for measuring inter-rater agreement has been extensively documented (see e.g. (Feinstein and Cicchetti, 1990; Feng, 2012; Lombard, 2004)). The main criticism relates to its sensitivity to prevalence and bias in the data, which also affects Krippendorff's alpha (Feng, 2012). While Cohen's kappa was used in by Vetro et al. (2012), they also included analyses to evaluate the degree of prevalence and bias, and were thereby able to evaluate the reliability of the kappa coefficient. According to Gwet (2008) and Feng (2012), a more reliable statistic for evaluating inter-rater agreement might be Gwet's AC_1 although it too has been shown to be affected by prevalence (Feng, 2012). Feng, furthermore, describes a method to evaluate the degree of prevalence and its implications on the choice of analysis method (Feng, 2012). Taken together, this shows that existing methods to evaluate consistency are not sufficiently reliable, and conclusions based on these methods should be taken with care. Still, as consistency is an important

[19] Note that, while this refers to the individual that performs the rating, in the statistics literature 'subject' refers to the objects being rated and the 'rater' to the individual who is performing the rating

aspect of assessing reliability of classification data, one interesting way of characterizing the inter-rater agreement is by using bubble diagrams as shown in Figure 26 and Figure 27.

Finally, while the use of university students as subjects is disputed in terms of external validity, it provides a convenient sample to acquire quantitative data. In our experiment, we applied blocking on amount of experience and acquired skill to evaluate its impact on classification performance. Extrapolating the differences provided means to reason about the generalizability of the results to an industrial setting.

8.7 CONCLUSION

In this paper, we have reported on a two-part study aimed at providing a comprehensive evaluation of defect classification schemes. In the first part of the study, two classification schemes—on generic scheme (ODC) and one adapted to the specific context under study (LiDeC)—were compared using a controlled experiment with university students as subjects. The experiment examined the two DCSs from three perspectives: learning and classification time, accuracy and consistency. Additionally, as comprehensive evaluations of DCSs have received little previous research attention, we have provided an in-depth description of and reflection on methodology.

In the second part of the study, we reported on an industrial case study in which LiDeC was applied to defects from a project. The aim of the study was to evaluate how the classification data could be shown useful in an industrial context. The results showed that the analysis of classification data could provide new information. In our study, that information raised interest at our case company and inspired discussions and critical evaluation of the development process.

Put together, the results from these two studies have provided evidence of defect classification as an efficient and effective approach to collecting data, which in turn can provide important insights into process and product quality.

8.8 ACKNOWLEDGEMENTS

The research presented in this paper is partially sponsored by VINNOVA under the V-ICT program in the ASIS (Algorithms and Software for Improved Safety) project. We would like to thank the people at Volvo Car Corporation who kindly took time out of their daily duties to participate in the case study. We would also like to thank all the students at Software Engineering and Management programme who participated in the experiment. Finally, we are very grateful for all the tips and ideas regarding the data analysis provided by Richard Torkar, Vera Lisovskaja, and Robert Feldt.

V

PART V—FUTURE DIRECTION

Plans are of little importance, but planning is essential
— Winston Churchill

9 FURTHERING THE USEFULNESS OF DEFECT CLASSIFICATION

The chapter examines current state-of-the-art defect classification and proposes a three-part research roadmap that is argued would advance the applicability of defect classification—both as a practical tool and as a research instrument. The chapter has previously been published as:

> Mellegård, N., Staron, M., Törner, F.
> *Why Do We not Learn from Defects?*
> *Towards Defect-Driven Software Process Improvement*
> Published at International Conference on Model-Driven Engineering and Software Development (ModelsWard), Barcelona Spain 2013

9.1 INTRODUCTION

Software defect classification schemes—such as ODC, HP and IEEE 1044—have the purpose of providing defect reports with a common structure. Such a structure allows for efficient quantitative analyses, which can provide evidence of the efficiency and effectiveness of various process activities. Following ODC, defect classification schemes (DCS) have been around for more than two decades—during which, software development has evolved

from being code- and document centric to be model-driven. Based on the number of publication in the area, however, we conjecture that DCS have had limited industrial impact—this limited impact is taken as a symptom of that the approach has failed to meet its target.

Despite limited adoption, publications—our own case study included—show that defect reports can provide valuable information for improving modelling when aggregated and analysed; to be an efficient tool to draw attention of various stakeholders to the most common, important or dangerous problems with software products.

We approach this apparent contradiction by addressing the question: "*As academic evidence show that DCS can be successful, why has it not had a more industrial impact and what can be done?"* We first address this question by concretely showing the value of DCS, using the synthesis of two industrial case studies from our previous work. We then provide evidence in support of the conjectured limited industrial adoption of DCS, and present reasons why. Finally, we provide a roadmap that would fill the gaps in current state-of-research that we envision would allow for a more successful approach to DCS.

In particular, we envision that the roadmap will contribute to making the results of defect analyses more useful to project stakeholders in control of resources, in particular in the system modelling phase. This is in contrast to the current state-of-the-art where analyses of classification data primary are intended for developers. By refocusing the DCS approach we envision that it will better serve as a tool for fact-based decisions during modelling—based on descriptive and predictive measures and indicators. The improvement would contribute to more accurate targeting of process improvement initiatives, to serve as the basis for defect-driven software improvement initiatives.

In practice, the purpose of a defect report is often limited to facilitating the resolution of the defect. For instance, defect reports are often free text (Wagner, 2008) which makes quantitative analyses effort intensive. In response, various DCS have been proposed. DCS also contribute to comparability of defect metrics between projects and between companies (Chillarege et al., 1992).

Classification schemes typically define a set of attributes, where each attribute captures a specific aspect of the defect—e.g. how the defect was detected, its severity and type. Each attribute typically contain a set of values that can be chosen from; this contributes to the efficiency and reliability of the classification. In literature, the most commonly referred (Freimut, 2001)

DCS are ODC (Orthogonal Defect Classification) (Chillarege et al., 1992), the HP approach (Grady, 1992) and the IEEE Std. 1044 (IEEE, 2009, 1993).

The attributes of ODC and IEEE Std. 1044 are organized into the defect's life-cycle phases; ODC, for instance, defines the phases open and close (shown in Table 31). The attributes in the opener section of ODC focus on aspects of the failure, whereas the closer section focuses on aspects of the fault.

Table 31 Overview of ODC (adapted from (Freimut 2001))

Process	*Attribute*	*Meaning*
Open	Activity	When did you detect the defect?
	Trigger	How did you detect the defect?
	Impact	What would the customer have noticed if the defect had escaped into the field?
Close	Target	What high level entity was fixed?
	Source	Who developed the target?
	Age	What is the history of the target?
	Type	What had to be fixed?
	Qualifier	Was the defect caused by something missing, incorrect or extraneous?

DCS typically focus on technical aspects of the defects and their source code manifestations; IEEE Std. 1044, for instance, lists 80 different values for its *Type* attribute.

9.2 IMPORTANCE OF DEFECT CLASSIFICATION

In our earlier work (Mellegård and Staron, 2010c; Mellegård, 2010) we investigated the importance of various artefact types in the automotive software development—such as requirements, types of software models, and documents. Specifically, we investigated the perceived importance of the artefacts and the relative effort required to create them. The particular focus of the case study was to characterize the use of software models in relation to other types of development artefacts.

Among the conclusions of the case study were that most effort was spent on simulation models (e.g. Simulink models), while the most important artefacts were the requirements and design artefacts. This result was in itself

not surprising as the simulation models serve as a base for the implementation, and as development is highly distributed—both among in-house teams and external suppliers—the quality of specifications is crucial to preventing eventual integration problems. In fact, during the case study we repeatedly encountered statements from our interviewees—expert opinions—that integration was a considerable challenge, in particular during the late project phases. Additionally, we frequently encountered concerns about lack of more objective evidence to support these expert opinions.

These findings directed our interest towards in-process defects, specifically to defects detected during late project phases; did integration issues cause these, and could we find evidence that the cause was as had been anticipated?

9.2.1 Cause of Late Defects

In a second study (chapter 7), we set out to make an in-depth examination of in-process defects from one system developed at the department. We found, in the initial analyses of the existing defect data, that there was indeed a substantial inflow of defects in late project phases; shown in Figure 31 as the defect backlog.

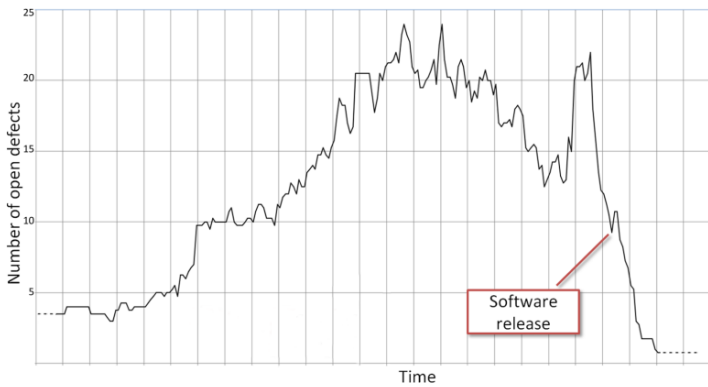

Figure 31 Defect backlog

The timing of the late spike in defects close to software release (a major in-development milestone) seemed to confirm the hypothesis of integration issues. Merely examining the quantity of defects, however, would not reveal the nature of the defects and as the defect reports were in free text, they were not suitable for quantitative analysis. We therefore used IEEE Std. 1044 as

base to develop a DCS adapted to the context of our case company. The result was the Light-Weight Defect Classification scheme (LiDeC) (chapter 6).

As part of the case study, LiDeC was applied to a sample of defects from the late defect inflow spike as shown in Figure 31 and reported in (Mellegård et al., 2012a), and the analysis results were presented and discussed at a workshop at the company.

Figure 32 shows a sample of the analysis using four attributes from LiDeC (see Appendix A. LiDeC Attribute Description), from three perspectives: detection of the failure, type of fault and finally product and project impact of the resolution. Using these three perspectives, the defects can be examined to evaluate whether they were integration issues as anticipated.

While the majority of defects was, as can be seen in Figure 32, indeed detected during integration testing—system or functional—the defect types were not typical for integration issues. Whereas the anticipated type would be *Interface, Data* or *Tuning*, the majority of defects were of the type *Logic*—i.e. computational or algorithmic faults. Such defects would normally be present already in the simulation models. This was corroborated by the *Resolution impact* attribute, shown in the bottom left of Figure 32, indicating that most defects required changes to a single unit—integration issues would typically have an impact on multiple units, or require changes to the specification (denoted *Functional changes*).

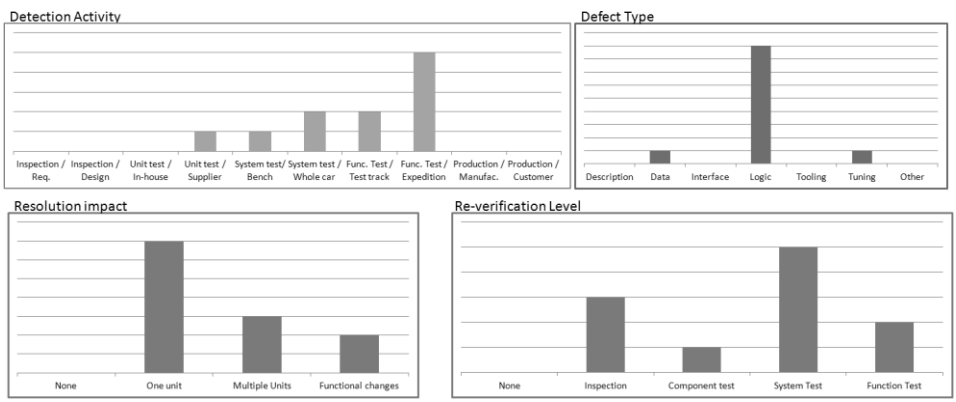

Figure 32 Preliminary analysis results

However, the *Re-verification Level* attribute—showing the activity required to test a resolution—indicates that it is not a clear-cut case. On the one hand, a significant amount of defects requires only inspection or

component test, indicating unit problems. On the other hand, most defects would require new system or functional tests, indeed indicating integration impact—finding the root cause of these defects could bring significant benefits.

Our (careful) conclusions from this study is that the majority of late defects—although to a large extent requiring new integration tests—are not of the type typically associated with integration problems. Thus, the classification of defects provided the development teams with new information that may contribute to better test planning—e.g. put effort into improving testability of requirements on unit level.

Finally, we would like to emphasize that the analysis presented here was conducted on a sample of defects from a project that had finished a year prior to the study. The results should therefore be treated as proof-of-concept rather than as a basis for recommended change of practice. We can however conclude that the classification contributed with new information that in part contradicted expert opinion. This raised interest and inspired discussions regarding possible causes; the case study provides evidence that conducting defect classification and analysis contributes to constructive review of the state-of-practice.

9.3 Problem Description

As we illustrated above, the use of DCS can be an effective approach to extracting data from problem reports, and can provide new information about the development process. In addition, there have been studies reporting similar results, e.g. Butcher et al. (Butcher et al., 2002), or Li et al. (Li et al., 2012).

However, we conjecture—partially based on our observations—that DCS has had limited industry adoption; we take this as a symptom that state-of-the-art DCS as a means of extracting process metrics has missed its target. To find evidence in support of this conjecture we searched IEEE Xplore (http://ieeexplore.ieee.org) using the search term: (‘defect classification’ AND ‘software’). The search, performed Oct. 25 2012, yielded 70 publications between 1986 and 2012. By reading titles and abstracts, we found that the publications fell into the following categories:

- *Proposing new DCS*, e.g. (Chillarege et al., 1992; IEEE, 2009; Paul et al., 2002)
- *Improving existing DCS*, e.g. by assisting a user in conducting the classification (Huang et al., 2011; Wang He et al., 2009), or adapting the scheme to a specific context (Li et al., 2010a)

- *Academic evaluation* of a DCS, e.g. (Henningsson and Wohlin, 2004; Vetro' et al., 2012)
- *Industrial evaluation* of a DCS, e.g. (Butcher et al., 2002; Chillarege and Ram Prasad, 2002; Freimut et al., 2005; Li et al., 2012)
- *Analysis techniques* for classification data (Li et al., 2010b)
- *Using classification data* for other purposes, e.g. to evaluate efficiency of inspections (Nagappan et al., 2004), to evaluate static analysis for fault detection (Zheng et al., 2006), to propose reliability estimation models (Paul et al., 2000), or to evaluate fault injection techniques (Jin and Jiang, 2009)

Notably, we found no publications evaluating industrial adoption of DCS, nor investigating what companies would require from such an approach; specifically in terms of the information that analyses of the data need to provide.

Further support for our conjecture can be found in the systematic literature review by Hall et al. (Hall et al., 2011). In their paper, Hall et al. examined 36 fault prediction models and noted that the vast majority of the models were limited to predicting the quantity of faults per module. In fact, Hall et al. could only find one model that incorporated fault severity as a predicted variable. In their paper, Hall et al. argue that one reason may lay in the difficulty of defining severity. An additional reason, however, may simply be a lack of available data; companies tend to collect only the defect data necessary to facilitate the resolution of the defect. This brings our thesis to a point: *although defect classification has been shown to be an effective approach to acquiring process metrics, why has it not had a wider industrial adoption?*

We can consider this point in the context of communication paths (Pareto et al., 2012). In their paper, Pareto et al. argue that a source of problems in projects is miscommunication because the needs and concerns of developers are not expressed in terms that managers and architects need in order to make informed decisions—rather developers express their needs in highly detailed and specific technical terms. In order to make an impact on the stakeholders in power, developers need to create abstractions suitable for that specific stakeholder; to provide evidence that level of abstraction.

In this context, we contend that established DCS are too focused on the developers' context—illustrated, for instance, by the high granularity of the *Type* attribute in both ODC and the IEEE 1044. In particular, established DCS fail at providing sufficient guidance to translate the results into the

language needed to make an impact on the stakeholders that are in control of the resources.

Furthermore, unnecessarily high level of detail in classification brings a risk that may have a double impact: on the one hand, it adds to the effort needed to make a classification (and thus reduces the available resources to resolve the problem). On the other hand, it adds to the required analysis effort needed to adapt results to the stakeholders in control. Consequently, effort risks being put into collecting data that remains unused (Li et al., 2012).

9.4 ROADMAP

The roadmap is divided into three parts: investigation of current DCS state-of-practice; investigation of how the design of DCS could meet the needs better; and finally, investigation of how to analyse the data to meet the organizational information needs.

9.4.1 State-of-practice

As we found a notable lack of research into current DCS state-of-practice in general and in modelling specifically, we envision surveying:

- To what extent do companies use DCS—in particular in the context of how DCS provides input to decision formulation and execution processes;
- What alternative approaches are used to facilitate analyses of defect reports on an aggregate level (e.g. none, defect taxonomies, root-cause analysis etc).

Furthermore, the results of the surveys should be correlated with aspects such as the size of the company and development teams, types of products developed and types of development processes used. Such an in-depth understanding of current practice would contribute to improving state-of-the-art DCS.

Additionally, there is a need to investigate the information needs of relevant project stakeholders—mainly product and project managers, and architects. In their paper Buse and Zimmerman (Buse and Zimmermann, 2011) examine general information needs among the stakeholders at a large software organization, exemplified by Microsoft. We, however, call for a more targeted investigation into which stakeholders are relevant, and what their information needs are, with the specific focus on defect data. We envision the needs falling into two main categories: descriptive and predictive.

Descriptive information would characterize project phases in terms of their defect profile (patterns). Descriptive information could be used in-process for benchmarking against a company base-line (Chillarege et al., 1992), for instance to provide evidence for evaluating process improvements.

Predictive information needs relate to the challenges of resource planning. For instance, in assigning resources of test phases in a project, there may be a need to predict the anticipated amount and type of defects—fault prediction models, such as reviewed in Hall et al. (Hall et al., 2011) aim at that. However, more granular defect data may enable more precise prediction models, thus enable defect-driven proactive decision support.

9.4.2 Design of DCS

The state-of-practice investigations should be complemented with establishing a library of best practices and lessons learnt in both the design and application of DCS. Li et al. (Li et al., 2012) made a recent contribution in part by reporting a number of lessons learnt when applying DCS in two organizations, however more studies are necessary.

Even though the design of DCS is already well represented in state-of-the-art, there are aspects that are not sufficiently developed. For instance, DCS should be refocused from aspects of the implementation (source code) to covering all project phases – in particular modelling. There is, additionally, a need to build abstraction mechanisms into the DCS in order to reduce the required analysis effort, and to improve the comparability of data between projects and organizations.

LiDeC (Mellegård et al., 2012a, 2012b) contributes to this for the automotive domain, but studies in other domains are needed. For instance, the attribute values in LiDeC are structured hierarchically, which is an inherent abstraction mechanism. This allows attributes to be extended with values at more detailed levels while retaining comparability at higher levels of abstraction.

In addition, the design of DCS should maintain reference to the ISO/IEC 15939 standard (ISO/IEC 15939, 2007) in order to facilitate integration with other measures (e.g. for the purpose of predictions).

9.4.3 Data analysis

The arguably most challenging aspect of DCS is in analysing of the data. The thesis put forward in this paper is that state-of-the-art DCS have failed partly due to insufficient analysis methods. We propose therefore the need for research into analysis and visualization methods that satisfy typical

information needs and attract attention to the most important defect patterns (as proposed in section 9.4.1). For instance, identifying product and project characteristics, such as change patterns in source code or software models (by inspecting versioning systems), that correlate with defect inflow profiles would enable defect inflow prediction models based on data mined from software repositories.

We envision an analysis reference manual that maps a stakeholder's information need with a set of best practices—for instance as a recommendation on which attributes to include in the analysis and how to visualize the data.

Moreover, we assert that reporting on industrial case studies where specific organizational problems have been addressed by analysis of defect classification data would be of valuable—the work by Li et al. (Li et al., 2012) contributes to this end.

9.5 CONCLUSION

In this paper, we have examined defect classification schemes as a tool for collecting process metrics in model based automotive software development projects. Specifically, we have critically examined the quality of state-of-the-art defect classification by investigating its industrial adoption. Our thesis was that defect classification has had limited industrial adoption which we have argued to be a symptom of knowledge gaps in state-of-the-art DCS.

One main reason for limited industrial adoption is—in our view—that state-of-the-art DCS are inadequate for their purpose. In particular, there is a too strong of a focus on low-level aspects of the implementation; i.e., a tool primarily intended for developers. DCS thus fail to address that project stakeholders in control of resources need information on a different level of abstraction to make informed decisions. This means that state-of-the-art classification approaches are poorly designed to produce the results that are needed in order to make an impact in an organization; thus the effort invested in collecting data risks being in vain, as a large potential of the data remain unused.

We have proposed a roadmap for an improved defect classification approach that would contribute towards developing new proactive evidence-based software process improvement strategies—defect-driven software process improvement. The roadmap includes: making a deeper investigation of the current adoption rate in industry; investigation of the typical information needs of the project stakeholders that have control over resources; investigation of how to design DCS to support multiple levels of

abstraction, and finally; to investigate methods of data analyses to accommodate the information needs of the various project stakeholders.

These actions will contribute to making DCS more appropriately adapted to organizations' needs. This in turn, we conjecture, will result in wider industrial adoption.

9.6 ACKNOWLEDGEMENTS

This research is partially sponsored by The Swedish Governmental Agency for Innovative Systems (VINNOVA) under the Intelligent Vehicle Safety Systems (IVSS) programme.

VI

PART VI—IN CONCLUSION

Finally, in conclusion, let me just say this.
— Peter Sellers

10 SUMMARY AND CONCLUSION

This thesis reports on an investigation of defect classification as a continuous in-process tool for extracting well-structured data about defects. Analysing such data can reveal interesting information regarding the development practices. For instance, patterns in the data may indicate systemic issues with the development process or its enactment.

The main contributions of the thesis, presented in chapters 5 – 9, are based on five academic publications. Three of these publications have been peer-reviewed (chapters 5, 7 and 9), one is a technical report (chapter 6) and one is submitted for publication (chapter 8):

- Chapter 5 describes an industrial case study in which the use of software models was investigated. The aim of the case study was to characterize model-related development artefacts compared to non-model ones. The characterization was done based on practitioners' perception of the cost and benefit of developing the various artefacts. The case study provided interesting insights into the complexity of automotive software development, and the need for objective data in order to identify improvement opportunities
- Chapter 6 provides an in-depth description of a defect classification scheme based on IEEE Std. 1044 and adapted to the development of automotive safety software. The resulting classification scheme, named LiDeC (Lightweight Defect Classification scheme), maintains compliance with the standard. Although the chapter is based on a

technical report, the main contributions of the report have previously been published and peer-reviewed (Mellegård et al., 2013, 2012a). The additional contributions (described in more detail in section 6.2.2 on page 58) of the technical report include:
  - a significantly more detailed background description and research motivation
  - a description of the method used for adapting the IEEE Std. 1044
  - a significantly more detailed description of LiDeC, including a comparison with and a mapping to IEEE Std. 1044
- Chapter 7 provides a description of an initial industrial evaluation of LiDeC, in which a sample of defects were classified and analysed
- Chapter 8 provides a description of a comprehensive evaluation of defect classification schemes. In addition, the chapter reports the results of applying that evaluation by comparing LiDeC to ODC
- Chapter 9 provides an investigation of the current state-of-the-art defect classification and identifies research areas that need to be addressed. As a result, the chapter describes a research roadmap that would further the usefulness of defect classification schemes in industry, as well as academia.

10.1 Summary of Results

The main research question in this thesis was:

> *How to improve defect management in automotive software development using structured defect documentation?*

This research question was broken down into four separate research questions, for which the main findings are summarized in the subsections below.

10.1.1 Characterizing Automotive Software Development

Chapter 5 in Part II of this thesis reported on an industrial case study in which the development of automotive software was characterized from a model-driven development perspective. The main research question addressed in Part II was:

> ***RQ 1*** *How important and effort intensive are software models in automotive software development, and what are the opportunities for improving the utilization of models?*

The objective of chapter 5 was to characterize the development of automotive safety features with respect to the use of software models. Specifically, the characterization was done by investigating how perceived development effort and importance were distributed between various model and non-model related artefacts in the development of software-based vehicle features. The investigation was conducted as an industrial case study in which project managers responsible for a software based safety and security related functions were interviewed.

The results of the study include that, while requirements were considered the most important type of artefact, executable models were the most effort intensive. Executable models are commonly used as in-house simulation models during development and by suppliers to generate production code. Even though these models were not considered as important, they constituted the single artefact that received the most effort—an appraised 24% to 55% of the total effort spent. Considering these models as the implementation (i.e. the actual product under development), it means that 45% to 76% of the effort is spent on overhead activities. This in turn suggests that there is room for improvement; the question is "*what to improve?*"

Although not surprising in itself, the finding that the most effort-intensive artefacts were not considered the most important, it highlights an important challenge. Requirements—in particular concerning communication interfaces and auxiliary components—are a prerequisite for early modularization. Such modularization allows individual development teams in a distributed development organization the peace of mind to focus on their specific part of the implementation. What is interesting, such requirements are typically specified early in the development cycle, but can often only be validated in late stages when the various components are mature enough to allow more extensive testing. This gap between specification and validation was perceived by the interviewees as one of the main development challenges, and that would typically manifest as late integration problems. Although

perceived as a major source of late issues, it was not clear precisely which types of requirements that caused the most sever issues, or what improvement initiatives to undertake to avoid such issues.

An additional finding was that the use of software models was fragmented. For instance, various modelling notations were used in parallel and interchangeably, and no controlled sequence of models or transformations was used. Such a sequence of transformations is a tenet of MDD's claim to improve efficiency, for instance in order to reduce the required manual labour for generating artefacts (such as code) or to automatically maintain traceability. This limited use of model transformation and automatic traceability, suggest that while models receive a substantial amount of effort, they are not used to their full potential (similar indications have been found in other organizations (Burden et al., 2014)). However, suggesting the introduction of model transformations as a development silver bullet would fail to recognize the complexity of automotive software development. More specifically, transitioning to a MDD paradigm has been shown to change substantially the way-of-working (Staron, 2008). Such substantial changes are not feasible in the automotive domain, as the development of a car involves the parallel development of a large number of interdependent heterogeneous components—mechanical, electrical and software-intensive parts. Instead, one approach to improve the efficiency of software development may be in line with how Selic (Selic, 2003) proposes adopting of a model-driven process; to introduce locally in projects tools that are most appropriate for the tasks at hand, and that addresses current development challenges. To identify objectively the tasks and challenges in need for improved practices in a complex development context, however, is not straightforward.

10.1.2 Adapting a Defect Classification Scheme

The concept of defect classification schemes was in this thesis investigated as means to identify tasks and challenges in need for improved practices. Although the IEEE Std. 1044 defines a standard structure for defect classification, it requires adaptation to be efficient in a specific development organization. Chapter 6 in Part III, presents an adaptation of the standard to the development of automotive software. The research question addressed in chapter 6 was:

RQ 2 *How to improve efficiency of applying IEEE Std. 1044 in model-based automotive software development?*

In chapter 6, the research question was addressed by investigating how IEEE Std. 1044 can be adapted to automotive software development. The specific properties of the automotive domain that motivates the adaptation include the strong reliance on supplier side implementation. This limits the usefulness of generic classification schemes, such as IEEE Std. 1044, as they tend to have a substantial focus on source code manifestations of defects. In addition, as defect classification does not directly contribute to the development of the end-product, it was considered important to adapt the classification scheme to minimize classification effort while still providing the additional benefits of characterizing the defects. More specifically, the research questions addressed in the chapter were:

L1 *In the context of automotive safety-feature development, where source-code is often not available, how can a standard defect classification scheme be suitably adapted?*

L2 *As defect classification may be considered an administrative task, how can the adaptation of a standard defect classification scheme be done to minimize required learning and classification time?*

As reported in chapter 6, L1 was addressed by:

- Shifting the focus of the classification scheme from detailed aspects of the fault and its resolution to aspects of the discovery of the defect. As the implementation is mainly done by suppliers, most in-house process improvement potential lies in more efficient defect discovery activities. LiDeC reflects this by providing more detailed attributes in the *Recognition* phase (e.g. *Detection activity*, *Urgency*, *Severity* and *Effect*), while granularity of the attributes in subsequent phases have been reduced; e.g. the *Type* attribute is less granular than in IEEE Std. 1044 and detailed aspects of the resolution (captured by the IEEE attribute *Resolution*) has been omitted;
- Adapting attributes for safety specific purposes. In LiDeC the attribute *Functional Safety Impact* (which maps to the IEEE attribute *Societal*) records whether a defect impacts ASIL-classified requirements (ISO/DIS, 2011). In addition, the values of the Effect attribute (which maps to the IEEE *Symptom* attribute) was adapted specifically to provide a high-level characterization of safety feature problems (e.g. unintentional activation of a feature).

L2 was addressed by:

- Raising the level of abstraction of attributes. For instance, by providing only higher-level categories as values for the Type attribute
- Providing more descriptive attribute values. For instance, for the *Severity* attribute, instead of values such as *Low*, *Medium* and *High* more descriptive values aligned with the company's vocabulary were chosen values
- Providing attribute descriptions and values phrased using the terminology of the company
- Providing a classification guide with a flow-chart structure, and including typical examples for each attribute value
- Streamlined the attributes by removed or redefining attributes that required insights that the typical reported might not have (project schedule and risk). Attributes were redefined with care to retain the intentions of the original attribute but measured in more specific engineering terms; for instance, whereas the IEEE Std. 1044 attribute *Project schedule* aimed at appraising the direct impact on the project plan, LiDeC instead appraises the estimated amount of re-verification (an activity that may have a large impact on the project schedule).

An additional contribution of chapter 6 is that the description of the adapted classification scheme, together with the mapping to the standard is intended to add to the generalizability of the data collected—i.e. to allow comparisons between organizations. This description aims to contribute to a baseline on how to adapt and document a classification scheme, in order for data collected from various organizations to be compared. Such comparisons may in turn contribute to a deeper understanding on issues common among organizations, and the effects of initiatives to resolve them.

10.1.3 Evaluating a Defect Classification Scheme

While the need to adapt generic defect classification schemes has previously been recognized, it has not been established how to evaluate the resulting classification scheme. In Part IV, two research questions relating to the evaluation of DCS were addressed:

RQ 3.1 *How feasible is LiDeC for model-based automotive software development?*

In chapter 7, a pilot study evaluating the applicability and usefulness of LiDeC was presented. The evaluation was conducted as an industrial case

study in which practitioners classified a sample of defects from a finished project.

This evaluation is of interest for two main reasons. Firstly, despite being published already in 1993, there are to date few experience reports evaluating the applicability of IEEE Std. 1044. Secondly, as LiDeC was adapted from the standard, it is of interest to evaluate what impact the adaptation has. More specifically, as the adaptation mainly consisted of raising the level of abstraction of the attributes, there is a risk that the lower granularity of the data impedes its usefulness.

The case study showed that even though the granularity of the attributes were lower, analyses of the data still contributed with new information about the development process. For instance, whereas the typical type of late defects was anticipated to be integration issues—such as timing of signals between software modules or mismatches in communication interfaces—the analysis indicated that it was instead algorithms and code logic defects. As these types of defect would typically be present already in the executable models, it suggested that more improved unit level tests or more advanced simulation environments would contribute to detecting a substantial quantity of late defects earlier in the process. While these results did not provide a specific course of action, they did provide the experts with evidence that can be considered more objective. Such evidence inspired discussions and critical evaluations of the development process.

Moreover, observations from the case study showed that the practitioners quickly understood the attributes in LiDeC and were able to efficiently perform classification—classification time stabilized around 5 – 10 minutes per defects already after classifying 3 – 4 defects (see Figure 30 on page 142). The understanding of LiDeC was, furthermore retained by the practitioners, as can be seen by the significantly shorter classification time for the initial defects in the second classification session (also shown in Figure 30). As the classification time included the time required to read and analyse the defect report, it can safely be assumed that when performed in-process—when the details about the defect are in fresh memory—the additional time required to perform the classification is small.

Thus, as the time required for learning and performing defect classifications according to LiDeC is short and analysing the classification data provide valuable information, the adaptation of IEEE Std. 1044 can be considered a feasible approach for extracting well-structured defect data.

RQ 3.2 *How to evaluate an adapted defect classification scheme?*

Chapter 8 adds to the initial evaluation, presented in chapter 7, by providing a detailed description of a comprehensive evaluation method for defect classification schemes. In addition to evaluating the effectiveness of a classification scheme (i.e. its usefulness, as presented in chapter 7), the method evaluates its efficiency. As reported in chapter 8, the evaluation, conducted as a controlled experiment, compares LiDeC with a baseline classification scheme (ODC was chosen for this experiment). In the experiment, university students participated as subjects—i.e. convenience sampling.

The efficiency aspect is in the proposed method evaluated from three perspectives: time, accuracy and, consistency. The time perspective includes the time required to learn the classification scheme, and the time required to perform a classification. The accuracy perspective represents the extent to which the subjects are able to assign the correct attribute values to a given defect report. Finally, the consistency perspective represents the classification agreement among the subjects for a given defect report. While the time perspective is important in assessing the resources required to classify defects, accuracy and consistency facilitate assessing the reliability of the data.

The results of the evaluation showed that DCSs are quick to learn, and the classification requires little time—the required classification time appeared to stabilize already after classifying 3 – 5 defects, approximately 5 – 10 minutes in the case study and under 5 minutes in the experiment. It should be noted that in the classification time is included the time required to read the defect report. In an industrial setting, this time would correspond to the time required to document the defect, an activity that is done regardless of whether classification is performed. The evaluation of the reliability of the classification data indicated that subjects, even when lacking domain-specific knowledge, are able to analyse the defect reports and arrive at rational classifications that were significantly better than random classifications. Furthermore, as shown in chapter 8, more experienced subjects, using the domain-specific classification scheme (LiDeC) performed classification that is more consistent. These results extrapolated to an industrial setting, where the classification is performed by experts, suggest that classification data would be reliable.

However, in light of experiences from the industrial case study, there are cases where classification requires subjective interpretation of the defect reports. Such interpretations may result in inconsistencies in the data—i.e.

that the classification may be dependent on who is performing the classification rather than on properties of the defect in question. One specific example was the LiDeC attribute *Injection Activity* that intends to capture the underlying reason why the defect was introduced into the system (similar to root cause analysis). In the case study, the information necessary to assign a value to this attribute was often found lacking in the reports. Although this information would, according to the case study subjects, be known at the time of reporting, it would still need expert judgement. This highlights the need for process support; for instance, regular reviews of sample classifications in order to establish and maintain organizational classification praxis.

10.1.4 Future Directions

In final part of this thesis, chapter 9 investigates areas in need of research with respect to defect classification schemes. The specific research question addressed in Part V is:

RQ 4 *What are the future research directions for improving defect classification?*

In chapter 9, the current state-of-the-art defect classification was investigated, and as a result of that investigation three research areas was described that would benefit from further investigation. Specifically, a literature review was conducted in order to—based on published scientific work—assessed the industrial adoption and the academic recognition of DCS.

The result of the literature review shows that there are few published studies reporting on the use of DCS. Specifically lacking are industrial studies on how classification data are used in organizations. In chapter 9, this lack of academic publications is taken as an indicator of limited industrial adoption. One main reason for limited industrial adoption is—in our view—that state-of-the-art DCS are inadequate for their purpose. In particular, there is a too strong of a focus on low-level aspects of the implementation; i.e., a tool primarily intended for developers. DCS thus fail to address that the project stakeholders in control of resources need information on a different level of abstraction in order to make informed decisions. This means that state-of-the-art classification approaches are poorly designed to produce the results that are needed in order to make an impact in an organization—thus the effort invested in collecting data risks being in vain, as a large potential of the data remain unused.

In addition, the academic recognition of DCS is demonstrated by its use as a tool for validation. In the literature review, cases were found where

researchers used DCS data to show effects of various improved practices. For instance, in their paper Nagappan et al. (2004) demonstrated which types of faults their proposed automated software inspection technique was able to identify. In this case, Nagappan et al. used defect classification as a means to generalize their results—an organization could compare the typical defect type distribution to examine whether that organization could benefit from using the proposed technique. In light of this example, defect classification data has interesting academic applications. Still, from the results of the literature review, it seems to have had limited adoption.

As an attempt to address this limited adoption, a research roadmap is proposed in chapter 9. The roadmap includes: making a deeper investigation of the current adoption rate in industry; investigation of the typical information needs of the project stakeholders that have control over resources; investigation of how to design DCS to support multiple levels of abstraction, and finally; to investigate methods of data analyses to accommodate the information needs of the various project stakeholders. In addition to contributing to making DCS more appropriately adapted to organizations' needs, it may also contribute to establishing a wider baseline with respect to defects. Such a baseline can be used by researchers to demonstrate effects of various improved practices, and by practitioners to evaluate whether such practices would address the challenges in their particular organization. This in turn, we conjecture, will result in a wider industrial adoption.

10.2 CONCLUSION

In this thesis, we have investigated defect classification schemes as a means for collecting structured defect data. More specifically, we have provided an experience report characterizing challenges in identifying improvement opportunities in automotive software development. These challenges exemplify information needs present in a complex software development organization, and furthermore highlights the need for a light-weight technique to extract data from defect reports—defects considered as symptoms of process issues, which in turn may constitute improvement opportunities.

In addition, we have reported on a novel classification scheme, adapted from a generic one to the development of automotive active safety software. The classification scheme, LiDeC, was described in detail and evaluated in an industrial context. We have also described a comprehensive method for the

evaluation of adapted classification schemes, with a focus on effectiveness and efficiency.

Finally, a research roadmap has been proposed in which identified current knowledge gaps with respect to defect classification will be closed. This, we envision, will lead to using defect classification data as a means to propose and evaluate various process improvement initiatives.

REFERENCES

Atkinson, C., Kuhne, T., 2003. Model-driven development: a metamodeling foundation. IEEE Software 20, 36–41.

AUTOSAR, 2011. AUTOSAR Technical Overview (R3.2 Rev 1) (Technical report No. 067).

Baker, P., Loh, S., Weil, F., 2005. Model-Driven Engineering in a Large Industrial Context — Motorola Case Study. In: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science. pp. 476–491.

Beizer, B., 1990. Software Testing Techniques, 2nd Edition, 2 Sub. ed. Intl Thomson Computer Pr (T).

Bennet, T.L., Wennberg, P.W., 2005. Eliminating embedded software defects prior to integration test. CrossTalk 18, 13–18.

Bijlsma, D., Ferreira, M.A., Luijten, B., Visser, J., 2011. Faster issue resolution with higher technical quality of software. Software Quality Journal 20.

Boehm, B.W., 1981. Software Engineering Economics, 1st ed. Prentice Hall.

Bollain, M., Garbajosa, J., 2009. A Metamodel for Defining Development Methodologies. In: Software and Data Technologies. pp. 414–425.

Brown, A.W., 2004a. An introduction to Model Driven Architecture - Part I: MDA and today's systems [WWW Document]. The Rational Edge. URL http://www.ibm.com/developerworks/rational/library/3100.html (accessed 2009-12-14).

Brown, A.W., 2004b. Model driven architecture: Principles and practice. Software and Systems Modeling 3, 314–327.

Broy, M., 2006. Challenges in automotive software engineering. In: Proceedings of the 28th International Conference on Software Engineering. ACM, Shanghai, China, pp. 33–42.

Burden, H., Heldal, R., Whittle, J., 2014. Comparing and Contrasting Mode-Driven Engineering at Three Large Companies. Submitted for publication (2013-10-23) in Workshop on Software Engineering in Practice (SEIP) at 36th International Conference on Software Engineering (ICSE) 2014, Hyderabad.

Buse, R.P.L., Zimmermann, T., 2011. Information Needs for Software Development Analytics (Technical report No. MSR-TR-2011-8), Microsoft technical report.

Butcher, M., Munro, H., Kratschmer, T., 2002. Improving software testing via ODC: Three case studies. IBM Systems Journal 41, 31 –44.

Cavalcanti, Y.C., Mota Silveira Neto, P.A., Lucrédio, D., Vale, T., Almeida, E.S., Lemos Meira, S.R., 2011. The bug report duplication problem: an exploratory study. Software Quality Journal.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., 1992. Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering 18, 943–956.

Chillarege, R., Inc, C., 2006. ODC- a 10x for Root Cause Analysis [WWW Document]. URL http://www.chillarege.com/articles/odc-10x (accessed 9.22.13).

Chillarege, R., Ram Prasad, K., 2002. Test and development process retrospective - a case study using ODC triggers. Presented at the International Conference on Dependable Systems and Networks, 2002. DSN 2002, pp. 669–678.

CMMI Product Team, 2010a. CMMI for Acquisition (Technical Report No. CMU/SEI-2010-TR-032). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

CMMI Product Team, 2010b. CMMI for Development (Technical Report No. CMU/SEI-2010-TR-033). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Das Beauftragte der Bundesregierung für Informationstechnik, 2009. Das V-Modell XT [WWW Document]. Das V-Modell XT. URL http://www.v-modell-xt.de/ (accessed 11.23.09).

Dingsøyr, T., 2005. Postmortem reviews: purpose and approaches in software engineering. Information and Software Technology 47, 293–303.

Dubey, A., 2012. Towards adopting ODC in automation application development projects. In: Proceedings of the 5th India Software Engineering Conference, ISEC ’12. ACM, New York, NY, USA, pp. 153–156.

Eklund, U., 2013. Engineering software for mass-produced embedded systems - Ways-of-working, architecture and ecosystems for innovation (Doctoral thesis). Chalmers University of Technology.

El Emam, K., Wieczorek, I., 1998. The repeatability of code defect classifications. Presented at the The Ninth International Symposium on Software Reliability Engineering, 1998. Proceedings, pp. 322–333.

Farkas, T., Neumann, C., Hinnerichs, A., 2009. An Integrative Approach for Embedded Software Design with UML and Simulink. In: 33rd Annual IEEE International Computer Software and Applications Conference, 2009. COMPSAC '09. pp. 516–521.

Feinstein, A.R., Cicchetti, D.V., 1990. High agreement but low Kappa: I. the problems of two paradoxes. Journal of Clinical Epidemiology 43, 543–549.

Feng, G.C., 2012. Factors affecting intercoder reliability: a Monte Carlo experiment. Qual Quant 2012, 1–24.

Freimut, B., 2001. Developing and using defect classification schemes (No. IESE- Report No. 072.01/E). Fraunhofer IESE.

Freimut, B., Denger, C., Ketterer, M., 2005. An industrial case study of implementing and validating defect classification for process improvement and quality management. Presented at the 11th IEEE International Symposium on Software Metrics, 2005, p. 10 pp.–19.

Galin, D., 2004. Software quality assurance. Pearson Education Limited, Harlow, England; New York.

Gamer, M., Lemon, J., Singh, I.F.P., 2012. irr: Various Coefficients of Interrater Reliability and Agreement. Computer program, available from: http://cran.r-project.org/web/packages/irr/

Giese, H., Neumann, S., Niggemann, O., Schätz, B., 2011. Model-Based Integration. In: Model-Based Engineering of Embedded Real-Time Systems, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 17–54.

Glass, R.L., 1994. The software-research crisis. IEEE Software 11, 42–47.

Gorschek, T., Wohlin, C., Carre, P., Larsson, S., 2006. A Model for Technology Transfer in Practice. IEEE Software 23, 88–95.

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

Grossman, M., Aronson, J.E., McCarthy, R.V., 2005. Does UML make the grade? Insights from the software development community. Information and Software Technology 47, 383–397.

Gruszczynski, M., 2013. kripp.boot [WWW Document]. GitHub. URL https://github.com/MikeGruz/kripp.boot (accessed 5.29.13).

Gwet, K.L., 2008. Computing inter-rater reliability and its variance in the presence of high agreement. British Journal of Mathematical and Statistical Psychology 61, 29–48.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A Systematic Review of Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering PP, 1276 – 1304.

Hallgren, K.A., 2012. Computing Inter-Rater Reliability for Observational Data: An Overview and Tutorial. Tutor Quant Methods Psychol 8, 23–34.

Hayes, A.F., Krippendorff, K., 2007. Answering the Call for a Standard Reliability Measure for Coding Data. Communication Methods and Measures 1, 77–89.

Heijstek, W., Chaudron, M.R.V., 2009. Empirical Investigations of Model Size, Complexity and Effort in a Large Scale, Distributed Model Driven Development Process. Presented at the 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. SEAA '09, pp. 113–120.

Henningsson, K., Wohlin, C., 2004. Assuring fault classification agreement - an empirical evaluation. Presented at the 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04, pp. 95 – 104.

Hristu-Varsakelis, D., Levine, W.S., 2005. Handbook of networked and embedded control systems. Birkhäuser, Boston.

Huang, L., Ng, V., Persing, I., Geng, R., Bai, X., Tian, J., 2011. AutoODC: Automated generation of Orthogonal Defect Classifications. Presented at the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2011, IEEE.

IBM, 2012. ODC | Overview [WWW Document]. Overview of ODC ver 5.11. URL http://www.research.ibm.com/softeng/ODC/DETODC.HTM#overview (accessed 9.22.12).

IEEE, 1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 1–84.

IEEE, 1993. IEEE Std. 1044-1993 Standard Classification for Software Anomalies.

IEEE, 1996. IEEE Std. 1044.1-1995 -- Guide to Classification for Software Anomalies.

IEEE, 2009. IEEE Std. 1044-2009. Standard Classification for Software Anomalies.

ISO, 2005. ISO 9000:2005 — Quality management systems - Fundamentals and vocabulary. International Organization for Standardization / Technical Committee 176 (ISO/TC 176), Geneva, Switzerland.

ISO/DIS, 2011. ISO/DIS 26262 - Road vehicles — Functional safety. International Organization for Standardization / Technical Committee 22 (ISO/TC 22), Geneva, Switzerland.

ISO/IEC, 2001. ISO/IEC 9126-1, Software Engineering - Product Quality.

ISO/IEC, 2004. ISO/IEC 15504 -- Information technology — Process assessment.

ISO/IEC, 2007. ISO/IEC 24744:2007, software Engineering -- Metamodel for Development Methodologies (Text).

ISO/IEC 15939, 2007. ISO/IEC 15939 - Systems and Software Engineering – Measurement Process.

ISO/IEC/IEEE, 2010. ISO/IEC/IEEE Systems and software engineering - Vocabulary (ISO/IEC 24765).

Ivarsson, M., 2010. Experience driven software process assessment and improvement (Doctoral thesis). Chalmers University of Technology.

Jin, A., Jiang, J., 2009. Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification. Presented at the 15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09, pp. 55 –62.

Kan, S.H., 1995. Metrics and Models in Software Quality Engineering, 1st ed. Addison-Wesley Professional.

Kent, S., 2002. Model Driven Engineering. In: Integrated Formal Methods. pp. 286–298.

Korhonen, K., 2013. Evaluating the impact of an agile transformation: a longitudinal case study in a distributed context. Software Quality Journal 21, 599–624.

Krippendorff, K., 2004. Content Analysis: An Introduction to Its Methodology. SAGE.

Landis, J.R., Koch, G.G., 1977. The Measurement of Observer Agreement for Categorical Data. Biometrics 33, 159–174.

Leszak, M., Perry, D.E., Stoll, D., 2000. A case study in root cause defect analysis. Presented at the 22nd International Conference on Software Engineering (ICSE), 2000, pp. 428–437.

Leszak, M., Perry, D.E., Stoll, D., 2002. Classification and evaluation of defects in a project retrospective. Journal of Systems and Software 61, 173–187.

Li, J., Stalhane, T., Conradi, R., Kristiansen, J.M.W., 2012. Enhancing Defect Tracking Systems to Facilitate Software Quality Improvement. IEEE Software 29, 59 –66.

Li, N., Li, Z., Sun, X., 2010a. Classification of Software Defect Detected by Black-Box Testing: An Empirical Study. Presented at the Second World Congress on Software Engineering (WCSE) 2010, pp. 234–240.

Li, N., Li, Z., Zhang, L., 2010b. Mining Frequent Patterns from Software Defect Repositories for Black-Box Testing. Presented at the 2nd International Workshop on Intelligent Systems and Applications (ISA), 2010, IEEE, pp. 1–4.

Liparas, D., Angelis, L., Feldt, R., 2011. Applying the Mahalanobis-Taguchi strategy for software defect diagnosis. Automated Software Engineering 19, 141–165.

Lombard, M., 2004. A Call for Standardization in Content Analysis Reliability. Human Communication Research 30, 434–437.

Ludewig, J., 2003. Models in software engineering – an introduction. Software and Systems Modeling 2, 5–14.

Mathworks, 2010. Simulink - Simulation and Model-Based Design [WWW Document]. http://www.mathworks.com/products/simulink/. URL http://www.mathworks.com/products/simulink/ (accessed 8.27.10).

Mellegård, N., 2010. Method and Tool Support for Automotive Software Engineering (Licentiate Thesis, No 74L). Chalmers University of Technology.

Mellegård, N., Staron, M., 2010a. Use of Models in Automotive Software Development: A Case Study. Presented at the The First Workshop on Model Based Engineering for Embedded Systems Design, Dresden, Germany.

Mellegård, N., Staron, M., 2010b. Distribution of Effort Among Software Development Artefacts: An Initial Case Study. Presented at the Exploring Modelling Methods for Systems Analysis and Design, Springer Berlin / Heidelberg, Hammamet, Tunisia.

Mellegård, N., Staron, M., 2010c. Characterizing Model Usage in Embedded Software Engineering: A Case Study. In: 8th Nordic Workshop on Model Driven Software Engineering (NW-MoDE). Copenhagen, Denmark.

Mellegård, N., Staron, M., Törner, F., 2012a. A Light-weight Defect Classification Scheme for Embedded Automotive Software and its Initial Evaluation. Presented at the IEEE International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, Tx USA.

Mellegård, N., Staron, M., Törner, F., 2012b. A Light-Weight Defect Classification Scheme for Embedded Automotive Software (Technical Report No. 2012:04, ISSN:1654-4870), Research Reports in Software Engineering and Management. Chalmers University of Technology, Göteborg.

Mellegård, N., Staron, M., Törner, F., 2013. Why Do We not Learn from Defects? Towards Defect-Driven Software Process Improvement. Presented at the MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, p. 6.

Mellor, S.J., Clark, A.N., Futagami, T., 2003. Model-driven development - Guest editor's introduction. Software, IEEE 20, 14–18.

Mohagheghi, P., Dehlen, V., 2008. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications, ECMDA-FA '08. Springer-Verlag, Berlin, Heidelberg, pp. 432–443.

Nagappan, N., Williams, L., Hudepohl, J., Snipes, W., Vouk, M., 2004. Preliminary results on using static analysis tools for software inspection. Presented at the 15th International Symposium on Software Reliability Engineering (ISSRE), 2004, pp. 429 – 439.

Niggemann, O., Stroop, J., 2008. Models for model's sake. Presented at the 30th International Conference on Software Engineering (ICSE), 2008, pp. 561–570.

Pareto, L., Sandberg, A.B., Eriksson, P., Ehnebom, S., 2012. Collaborative prioritization of architectural concerns. Journal of Systems and Software 85, 1971–1994.

Paul, R.A., Bastani, F., I-Ling Yen, Challagulla, V.U.., 2000. Defect-based reliability analysis for mission-critical software. Presented at the The 24th Annual International Computer Software and Applications Conference (COMPSAC), 2000, IEEE, pp. 439–444.

Paul, R.A., Bastani, F.B., Challagulla, V.U.B., Yen, I.-L., 2002. Software measurement data analysis using memory-based reasoning. Presented at the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2002., pp. 261 – 267.

Peldzius, S., Ragaisis, S., 2011. Comparison of maturity levels in CMMI-DEV and ISO/IEC 15504. In: Applications of Mathematics and Computer Engineering - American Conference on Applied Mathematics, AMERICAN-MATH'11,. Presented at the 5th WSEAS International Conference on Computer Engineering and Applications, CEA'11, pp. 117–122.

Pfleeger, S.L., 2001. Software Engineering: Theory and Practice, 2nd ed. Prentice Hall.

Plant, R.T., Tsoumpas, P., 1995. A survey of current practice in aerospace software development. Information and Software Technology 37, 623–636.

Ploski, J., Rohr, M., Schwenkenberg, P., Hasselbring, W., 2007. Research issues in software fault categorization. SIGSOFT Softw. Eng. Notes 32.

R Core Team, 2012. R: A Language and Environment for Statistical Computing. Vienna, Austria. Computer program, available from: http://www.R-project.org/

Rout, T.P., Tuffley, A., 2007. Harmonizing ISO/IEC 15504 and CMMI. Software Process: Improvement and Practice 12, 361–371.

SEI, 2013. Measurement & Analysis | Overview [WWW Document]. Measurement & Analysis | Overview. URL http://www.sei.cmu.edu/measurement/ (accessed 6.19.13).

Selic, B., 2003. The pragmatics of model-driven development. Software, IEEE 20, 19–25.

Shatnawi, R., Li, W., 2008. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. J. Syst. Softw. 81, 1868–1882.

Sim, J., Wright, C.C., 2005. The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. PHYS THER 85, 257–268.

SPSS, 2013. SPSS Homepage [WWW Document]. URL http://www.spss.com/ (accessed 1.9.09).

Staron, M., 2006. Adopting MDD in Industry - A Case Study at Two Companies. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (Eds.), ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, LNCS. Springer-Verlag, Genova, Italy, p. 57–72.

Staron, M., 2007. Using Experiments in Software Engineering as an Auxiliary Tool for Teaching – A Qualitative Evaluation from the Perspective of Students' Learn-ing Process. In: 27th International Conference on Software Engineering (ICSE). IEEE, Minneapolis, p. (accepted for publication).

Staron, M., 2008. Transitioning from code-centric to model-driven industrial projects – empirical studies in industry and academia. In: Model Driven Software Development: Integrating Quality Assurance. Information Science Reference, pp. 236–262.

The SPICE User Group, 2011. Automotive SPICE [WWW Document]. Automotive SPICE. URL http://www.automotivespice.com/ (accessed 6.20.11).

Thung, F., Lo, D., Jiang, L., 2012. Automatic defect categorization. In: Proceedings - Working Conference on Reverse Engineering, WCRE. pp. 205–214.

Wagner, S., 2008. Defect classification and defect types revisited. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08. ACM, New York, NY, USA, pp. 39–40.

Wang He, Wang Hao, Lin Zhiqing, 2009. Improving Classification Efficiency of Orthogonal Defect Classification via a Bayesian Network Approach. Presented at the International Conference on Computational Intelligence and Software Engineering (CiSE), 2009, pp. 1–4.

Weigert, T., Weil, F., Marth, K., Baker, P., Jervis, C., Dietz, P., Gui, Y., van den Berg, A., Fleer, K., Nelson, D., Wells, M., Mastenbrook, B., 2007. Experiences in Deploying Model-Driven Engineering. In: SDL 2007: Design for Dependable Systems.

Vetro', A., Zazworka, N., Seaman, C., Shull, F., 2012. Using the ISO/IEC 9126 product quality model to classify defects: A controlled experiment. In: 16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012). Presented at the 16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012), pp. 187 –196.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslèn, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publisher, Boston MA.

Wood, A., 1996. Software reliability growth models (Technical Report No. 96.1). Tandem Computers Inc.,.

Yin, R.K., 2002. Case Study Research: Design and Methods. SAGE Publications.

Yin, R.K., 2009. Case Study Research: Design and Methods, 4. ed. ed, Applied social research methods series. SAGE, London.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A., 2006. On the value of static analysis for fault detection in software. IEEE Transactions on Software Engineering 32, 240 – 253.

A

Appendix A. LiDeC Attribute Description

This appendix provides a detailed description of the attributes and each attribute value in LiDeC.

Life-cycle phase 1 – Recognition

Table 32 LiDeC Scheme – Attributes in the Recognition phase

Attribute	Attribute Description	Values	Value Description
Timing/Detection *[RTD]*	When was the defect discovered?	Date *[RTD1]*	Date/project phase of detection
Timing/Preferred *[RTP]*	Was the defect discovered in the proper test phase according to the goals of the phase?	Yes *[RTP1]*	The discovery was timely; there was no previous test phase in which the specified goal included detection of this type of defect
		No *[RTP2]*	This value is only provided if it is apparent that the defect should have been caught in an earlier test phase. Also specify which test phase, e.g. *E1-Ex, M, VP, TT*
Affect SW *[RAS]*	Does the defect affect software? "Affect" is used here to denote either defects that are caused by anomalies in the software or whose resolution have an impact on software – an example of the latter may be that dirt causes a sensor to degrade, but if the system fails to detect this (through diagnostic software) and notify the driver of degraded performance, it still affects software. This attribute will be used as a way to filter the defect reports as we are mainly interested in defects that are either caused by software or where the resolution may affect the software	Yes *[RAS1]*	The defect affects software
		No *[RAS2]*	The defect has no relationship with software at all, e.g. testing of vibration resilience of the physical component failed

Attribute	Attribute Description	Values	Value Description
Detection activity *[RDA]*	What was done when the defect was discovered? Detection activity captures the type test action that was conducted to detect the defect.	Inspection / Requirements *[RDA11]*	The defect was detected during inspection of requirements specification
		Inspection / Design *[RDA12]*	The defect was detected during inspection of design specification, e.g FMEA
		Component test / VCC *[RDA21]*	The defect was detected while running a unit test in-house, e.g. unit testing of an isolated component using a SimuLink model.
		Component test / Supplier *[RDA22]*	D.o but the defect was detected by the supplier
		System test / System-bench *[RDA31]*	The defect was detected while running an integration test (multiple cooperating components realizing functionality) on a simulation of the target platform done in-house. As “simulation” of the target platform are considered “box-car” (early E-series) as well as “mules” (M-series)
		System test / Whole car-bench *[RDA32]*	This refers to system testing done on system simulation on rigs with the whole electrical system present; though not necessarily with the final hardware present E.g bench tests with the whole electrical system
		Functional test / Test track *[RDA41]*	Functional testing of system using a car-build – a mule of a test build of the final hardware. Test-track refers to testing isolated scenarios on a test-track i.e. with real sensor input, but a simulated test-setup (e.g. with balloon cars or dummies)

Attribute	Attribute Description	Values	Value Description
		Functional test / Expedition *[RDA42]*	Functional testing of the whole car using a test-build – mule or the final hardware build. The defect was detected on an expedition with real data
		Production platform / Manufacturing *[RDA51]*	Defect was detected during manufacturing, e.g. calibration or configuration of a function, such as calibration of sensors in the production line
		Production platform / Customer reported *[RDA52]*	The defect was detected by customer post-release, e.g. car owner or maintenance staff
Urgency *[RU]*	How urgently does the defect need to be addressed? Denotes how urgent the defect needs to be removed from the product – thus urgency is related to the *project*. In late stages of the project defect will naturally be more urgent than in earlier stages. However, defects in early project phases that are blockers (i.e. blocking other functionality from being testable), should be considered urgent	Immediate *[RU1]*	The defect should be removed in the current development cycle; i.e. before the next development release (e.g. detected in E3.1 and should be removed in E3.2). E.g. defects that are blocking vital functionality should be classified as "immediate" (as they are inhibiting testing of that functionality)
		Next major release *[RU2]*	The defect should be removed before the next major development release. E.g. detected in E3.x and should be removed by E4.
		Before SoP *[RU3]*	The defect should be removed before the software is released, i.e. SoP (Start of Production). E.g. minor flaws or functionality that can be adjusted using tuneable parameters, or documentation issues

Attribute	Attribute Description	Values	Value Description
		Deferrable *[RU4]*	Defects that are not considered to have much of an impact on product, and can be deferred until later versions or revisions of the product
Severity *[RS]*	How severely does the defect affect the product? Severity denotes the end-user perceived impact on the product if the defect is left in the released product – thus severity is related to the *product (i.e. vehicle)* not the *project*. Note, this attribute shall not consider the timing of the defect detection, i.e. regardless of when a defect is detected during the project it shall receive the same severity (whereas its *Urgency* may vary). Also note that *end-user* refer to the intended target of the feature, e.g. vehicle occupants as well as manufacturing and maintenance personnel.	None *[RS1]*	The defect would not be noticed by the end-user
		Nuisance *[RS2]*	The defect would be limited to a nuisance for the end-user – though the product would still realize the full functional specification. E.g. a warning system would still be able to function in all scenarios originally specified, but may give an increase amount of false warnings
		Limited functionality *[RS3]*	The defect would limit the functionality of the product – e.g. the product would still function but not to the extent originally specified.
		Show-stopper *[RS4]*	A "show-stopping defect" is one that would prevent the product from being released; e.g. defect that would result in increased risk of injury, or that block other functions from performing according to specifications.
Effect *[RE]*	How does the defect primarily affect the product? Note that these may be overlapping to some extent but classification should be done on the effect, not the cause (as life-	Capability / Undesired activation *[RE11]*	The defect causes the function to trigger on a false positive

Attribute	Attribute Description	Values	Value Description
	cycle phase 1 is focussed on the detection of defects) – e.g. low performance of the software in a sensor may affect the functionality of the system, thus defect should be classified as 'Functionality'	Capability / Inactive despite True Positive *[RE12]*	The defect inhibits activation of functionality despite presence of a true positive
		Capability / Other capability related defect *[RE13]*	The defect affects the capability of the product; i.e. the product does not behave as intended or to the extent intended
		Maintainability *[RE4]*	The defect would affect the maintainability of the system; e.g. documentation issues, too complex design, cryptic internal error codes, wrong or missing diagnostic codes.
		Usability *[RE5]*	The defect affects the systems ease of use; e.g. complex user interface, missing or wrong visual cues to driver
		Configurability *[RE6]*	The defect affects configuration or calibration of the function; e.g. configuration of vehicle model variations or calibration of components during manufacturing
		Testability *[RE7]*	The defect affects the testability of the product; e.g. radar software the fails to detect a balloon car at the test site
Functional Safety Impact *[RFS]*	Does the defect have an impact on a software component with ASIL-classified requirements (ISO 26262)?	Yes *[RFS1]*	The defect have an impact on ASIL-classified requirements according to the ISO 26262 standard (ISO/DIS, 2011).
		No *[RFS2]*	The defect does not affect ASIL-classified requirements.

Life-cycle phase 2 – Analysis

Table 33 LiDeC Scheme – Attributes in the Analysis Phase

Attribute	Attribute Description	Values	Value Description
Artefact [AA]	Which software work product contained the defect? This attribute relates to the work product in which the fault causing the failure was contained. Note that the underlying reason for introducing the fault may lie in another work product (see Injection activity)	Requirement / Internal [AA11]	Defect was contained in a requirement for the module itself
		Requirements / Internal cross-function *[AA12]*	The defect was contained in a requirement the module posed on an external module. E.g wrong required resolution posed on a sensor which is not part of the module itself.
		Requirements / External *[AA13]*	The defect was contained in a requirement posed on the module by another module. E.g an external module required wrong resolution of a sensor which is part of the module itself.
		Design model *[AA2]*	The defect was contained in a design model, e.g logical design (class diagram)
		Implementation / Executable model *[AA31]*	The defect was contained in a simulation model, e.g. Simulink
		Implementation / Code *[AA32]*	The defect was contained in code, either written in-house or by supplier, or code generated from models. Note, if code was correctly generated from a defective model, the defect should be classified as *[AA31]*
		Configuration Parameters *[AA33]*	The defect was contained in the tuning parameters for the function

Attribute	Attribute Description	Values	Value Description
		Tool *[AA4]*	Defect was contained in tools used during development; e.g. simulation environments for sensors
Injection activity *[AI]*	In which project activity was the defect injected? This attribute shall capture the reason why the defect was contained in the work product (as specified in the *Artefact* attribute); i.e. what caused the defect to have been introduced in the system. Note, it may differ from the artefact the defect was contained in , e.g. a design defect may have been injected due to a poor or missing requirements (artefact='Design model', Injection='Requirement')	Specification *[AI1]*	The defect was injected in the requirements phase; e.g. a missing, faulty, misrepresented, ambiguous requirement caused the defect
		Design *[AI2]*	The defect was injected in the design phase even though the requirements were stated correctly; e.g. missing or faulty signal between modules, or problems with the modularization
		Implementation / In-house model *[AI31]*	The defect was injected when constructing the simulation model in-house. Requirements and design were correctly specified, but mistake was made in an implementation model.
		Implementation / Supplier Auto-coding *[AI32]*	The defect was injected when transforming an executable model into code by supplier. Specification and simulation model were correct, but mistake was made in code generation by the supplier.
		Implementation / Supplier implementation *[AI33]*	The defect was injected in implementation at the supplier side (code not generated from simulation model by VCC); implementation based on correct specification and design from VCC
		Configuration *[AI4]*	The defect was injected in the configuration of the function (specification, design and implementation is correct); i.e. a faulty value of a tuning parameter

Attribute	Attribute Description	Values	Value Description
Component /Asset *[AC]*	Which design component contained the defect? This attribute shall identify the component (at code level) at the lowest level available. The purpose is the attribute is to identify which components are most likely to contain defects.	Component name or ID *[AC1]*	A unique ID of the component containing the defect. The higher the resolution the better
Type *[AT]*	What type of defect was it? The type attribute describes the character of the defect. The values of the type attribute may depend on which artefact/component it affects	Data *[AT1]*	Defect in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation (IEEE, 1993). E.g. initialization of a variable, incorrect assignment of a value, incorrect cardinality in data model, using wrong variable, assuming wrong variable type (e.g. assuming vehicle speed in km/h when it is stored as mph)
		Interface / Timing *[AT2]*	Defect in specification or implementation of an interface between two design components, e.g. missing or wrong signals specified or errors in the timing of communication
		Tooling *[AT4]*	The defect is present in tools used in development; e.g. simulation environments that are used in development (e.g. simulating external components such as sensors etc.)
		Logic / Computation *[AT3]*	A defect in the logic of execution; eg. an algorithmic defect either because of a faulty implementation of a correct specification or a faulty specification (or any combination thereof)

Attribute	Attribute Description	Values	Value Description
		Tuning *[AT5]*	The defect relate to tuning parameters of the function. E.g. missing or misinterpreted tuning parameters (errors in design) or faulty values assigned to tuning parameters (implementation)
		Description *[AT6]*	Defect in specification, e.g. missing or wrong description (such as a requirement)
		Standards *[AT7]*	Non-conformity with a defined standard

Life-cycle phase 3 – Resolution

Table 34 LiDeC Scheme -- Attributes in the Resolution Phase

Attribute	Attribute Description	Values	Value Description
Timing/Removal *[ET]*	When was the defect removed? Note, this does not necessarily mean that the defect was fixed (see attribute 'Resolution state' in life-cycle phase 4)	Date / Project phase *[ET1]*	The date/project phase when the defect was considered removed from the system, i.e. when the defect report was closed.
		Defect not yet closed *[ET2]*	The defect has not yet been closed. NOTE! This refers to the defect/problem report not the fault or failure; i.e. a defect report can be closed without having the underlying fault addressed.
Product Impact *[EI]*	What is/would be the impact on the product of a proper resolution? Note, this is an estimate of a *proper* resolution of the defect; i.e. an issue that would require major redesign to resolve, but that can be worked around with a small local fix shall be classified as a "Re-design' (the scope of the change made is captured by 'Resolution state' in life-cycle phase 4)	None *[EI1]*	There is no resolution (e.g. the reported defect was intended behaviour) or the resolution has no impact on the product
		Local modification *[EI2]*	The resolution is limited to a fixing a local module; other modules are not affected E.g. modification of a tuning parameter or code modifications to a single module that does not affect other modules
		Multiple components *[EI3]*	The resolution requires changes in multiple existing modules
		Functional Changes *[EI4]*	The resolution require a redesign, e.g. adding, removing or redefining modules
Required Verification Level *[EV]*	What level of regression testing would a proper resolution require?	None *[EV5]*	No re-verification needed

Attribute	Attribute Description	Values	Value Description
		Inspection *[EV1]*	Review of documentation, report or code is sufficient means of verification/validation of the modified component/system
		Component test *[EV2]*	Re-verification of the modified component using recorded data (Resim) is sufficient. E.g. running an executable model in Simulink with the recorded data
		System test *[EV3]*	Re-verification at system level using recorded data (Resim) is sufficient E.g. a new software build of all components of the system is needed in order to validate the changes; it is sufficient to re-verify the system on bench with recorded data (Resim)
		Expedition *[EV4]*	The resolution would need re-validation with real data, e.g. in a full car-build on test-site or on an expedition (“Fälttest”, “Breddprovning”)

Life-cycle phase 4 – Post-mortem

Table 35 LiDeC Scheme -- Attributes in the Post-mortem Phase

Attribute	Attribute Description	Values	Value Description
Resolution state [PS]	What was the state of the resolution when the defect was marked as closed? This attribute is meant to track how the defect was eventually handled	Corrected [PS1]	A proper resolution, addressing the root cause, was applied
		Workaround / Fix *[PS21]*	The underlying fault remains, but workarounds were made to avoid failure. The workaround retains the intended capability of the original specification
		Workaround / Product de-scope *[PS22]*	D.o. but the workaround forced de-scoping of the system, e.g. limiting the functionality or quality of service
		No Action / Deferred *[PS31]*	The defect was left in the system, and resolution deferred to a later revision
		No Action / Referred *[PS32]*	The source of the defect lies in another system. Defect was referred and closed in this system
		No Action / Not Found *[PS33]*	The defect was not found again; e.g. the failure could not be reproduced or the defect was not observed in a later revision of the software
		No Action / No Action *[PS34]*	No action taken, defect remains in system

B

Appendix B. LiDeC Classification Guide

Figure 33 to Figure 45 show the classification guide used during the classification sessions, as described in section 6.3 *Method*. Initially, LiDeC was presented to the person responsible for defect classification by an overview of the classification scheme. The overview was described by using Figure 33, in which all available LiDeC attributes are represented and grouped into the phases of the defect life-cycle (further described in *Defect classification schemes* on page 62).

During the classification sessions, each defect was classified according to the attributes in LiDeC. For each attribute, the developer was shown the corresponding image (shown below). In each image, the question that guides the reporter is shown at the top in an orange rectangle (e.g. "What was done when the defect was detected?"). The values that can be assigned the attribute are shown in blue rectangles below the question. The white boxes, shown e.g. in the attribute "Detection Activity", represent categories of values and serve only to provide the values with a clearer structure. For instance, for the attribute "Detection activity" the categories serve to make an initial separation on types of activities, and then breaks down those into the sub-activities that are to be chosen as the value for the attribute.

Note, in this appendix the guides for trivial attributes (i.e. attributes with simple Yes/No values or dates) have been omitted.

Attribute overview

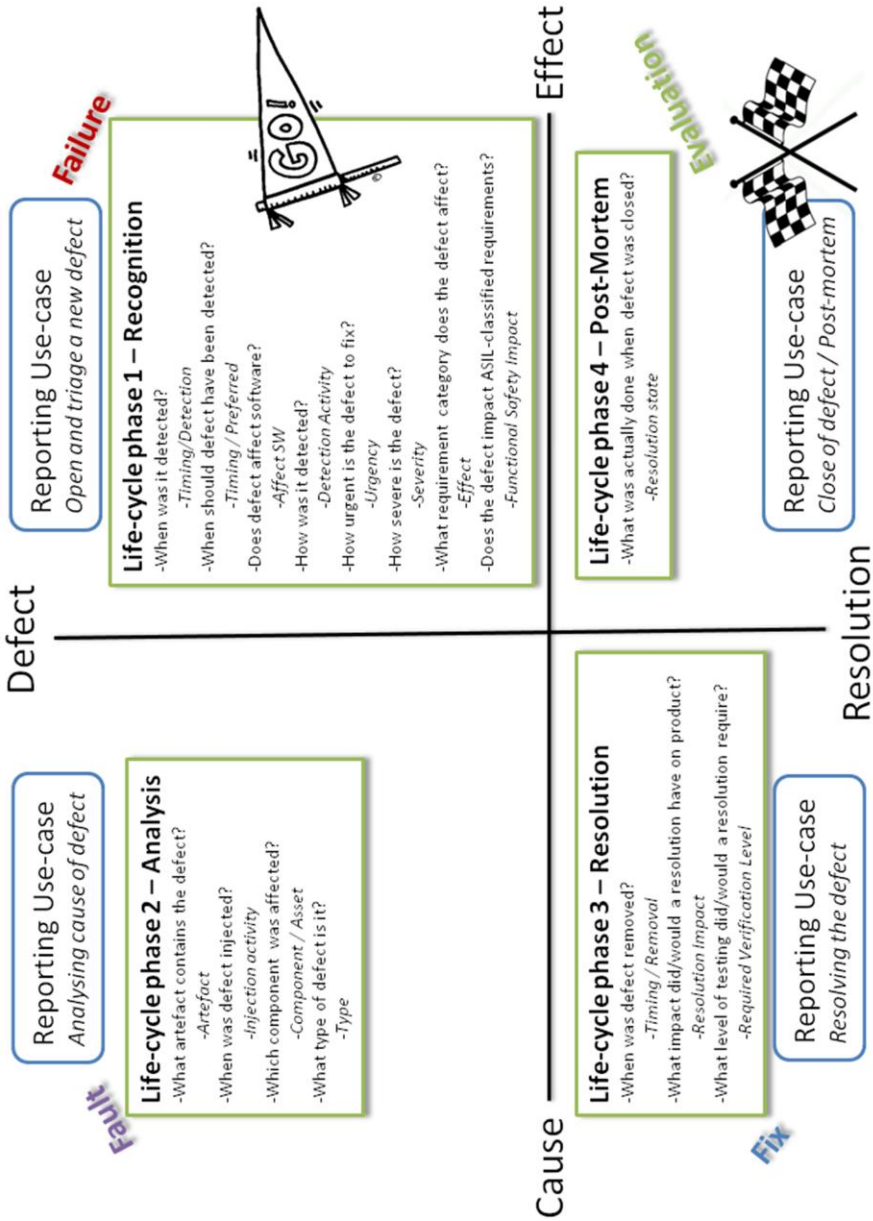

Figure 33 Overview of LiDeC attributes

Life-cycle phase 1: Recognition – Detection of the defect

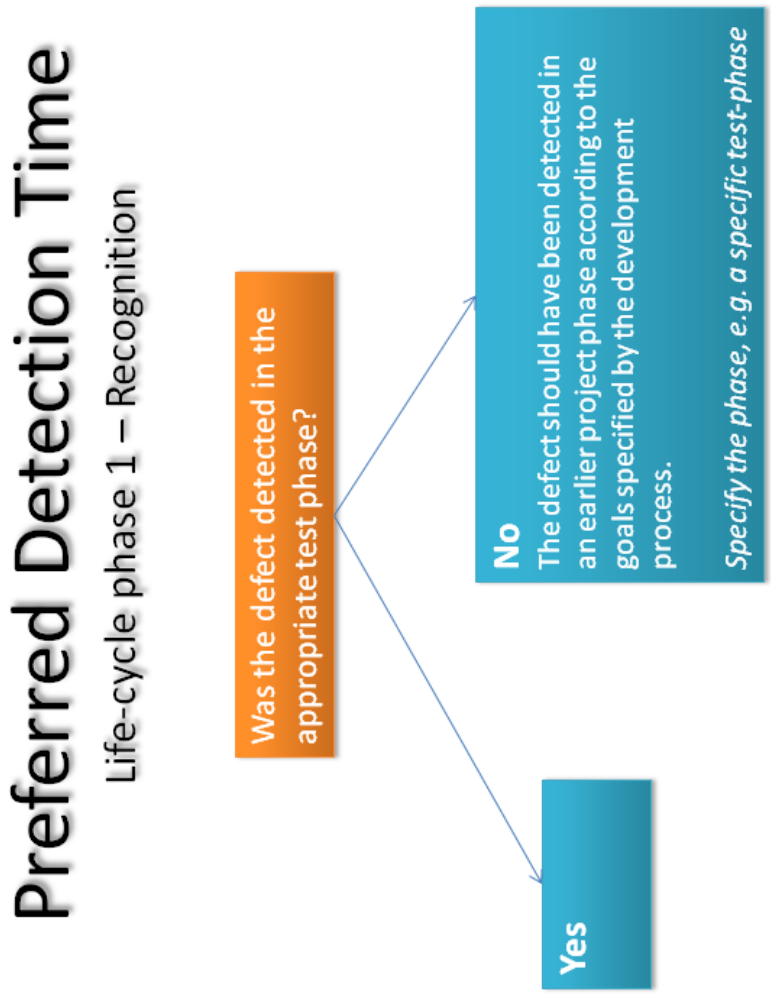

Figure 34 Classification guide for the attribute Preferred detection time

Figure 35 Classification guide for the attribute Affect Software

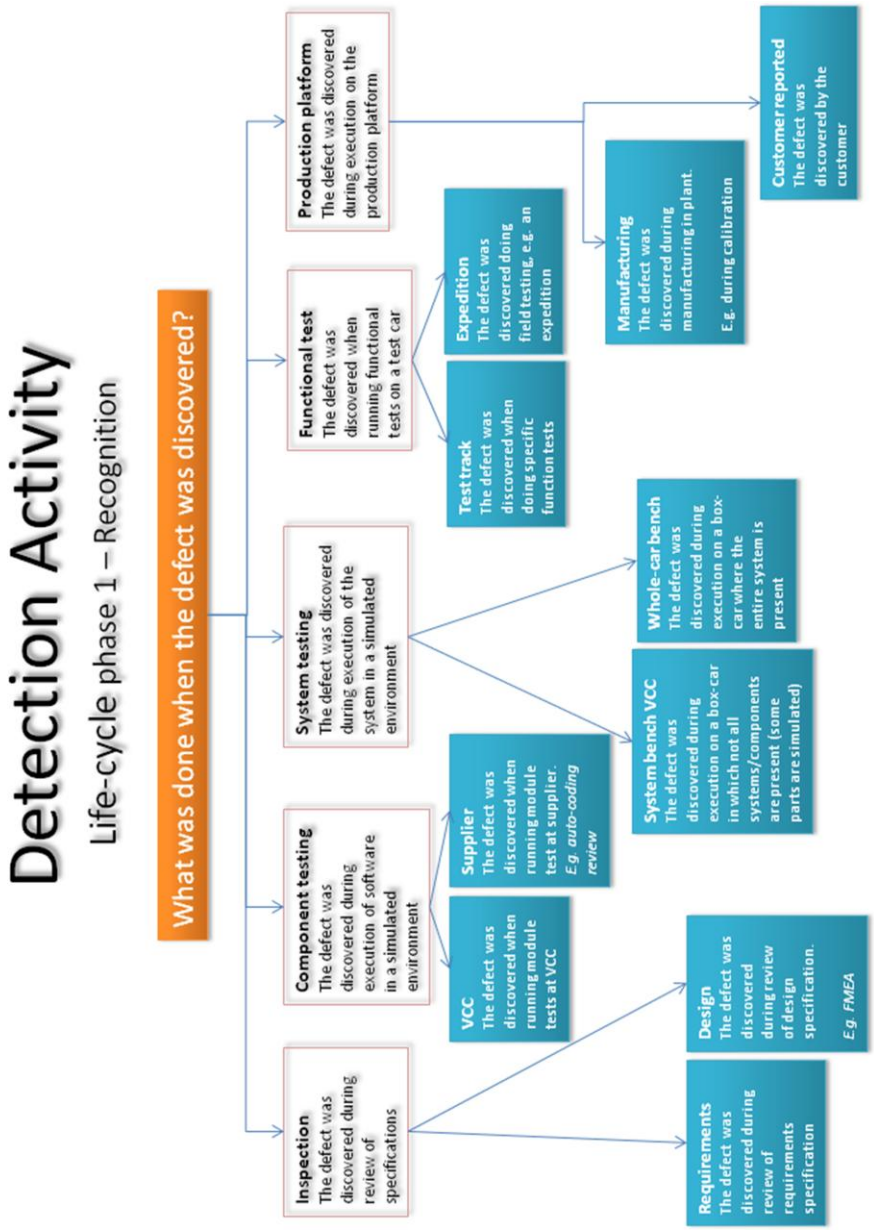

Figure 36 Classification guide for the attribute Detection activity

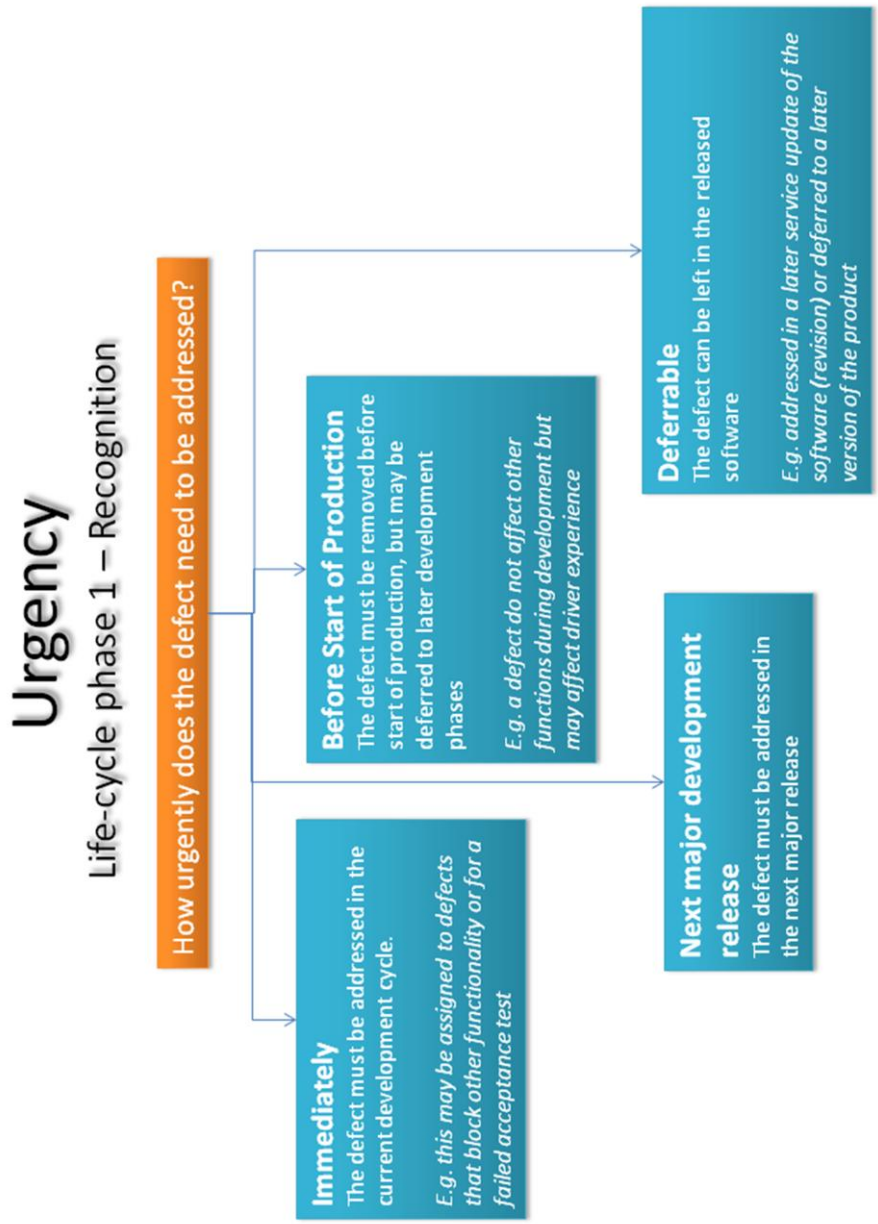

Figure 37 Classification guide for the attribute Urgency

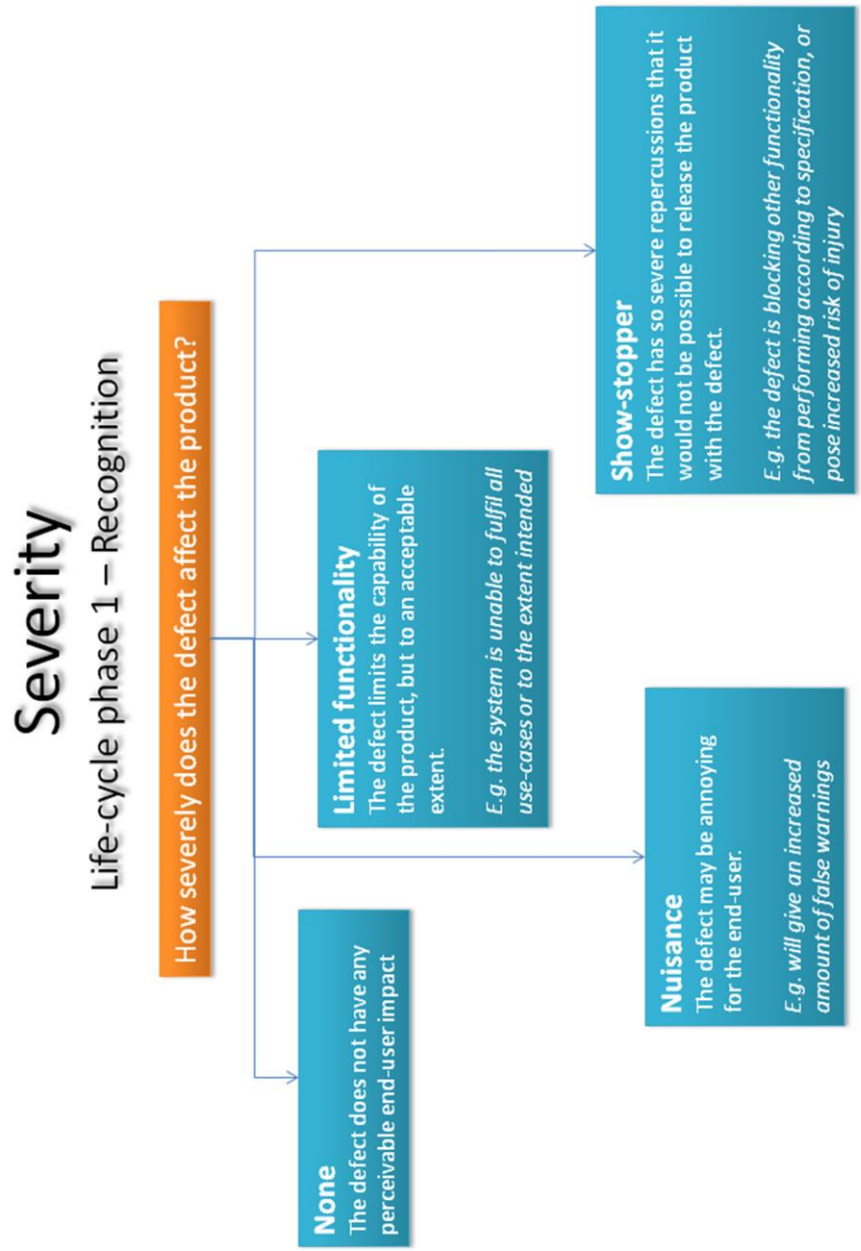

Figure 38. Classification guide for the attribute Severity

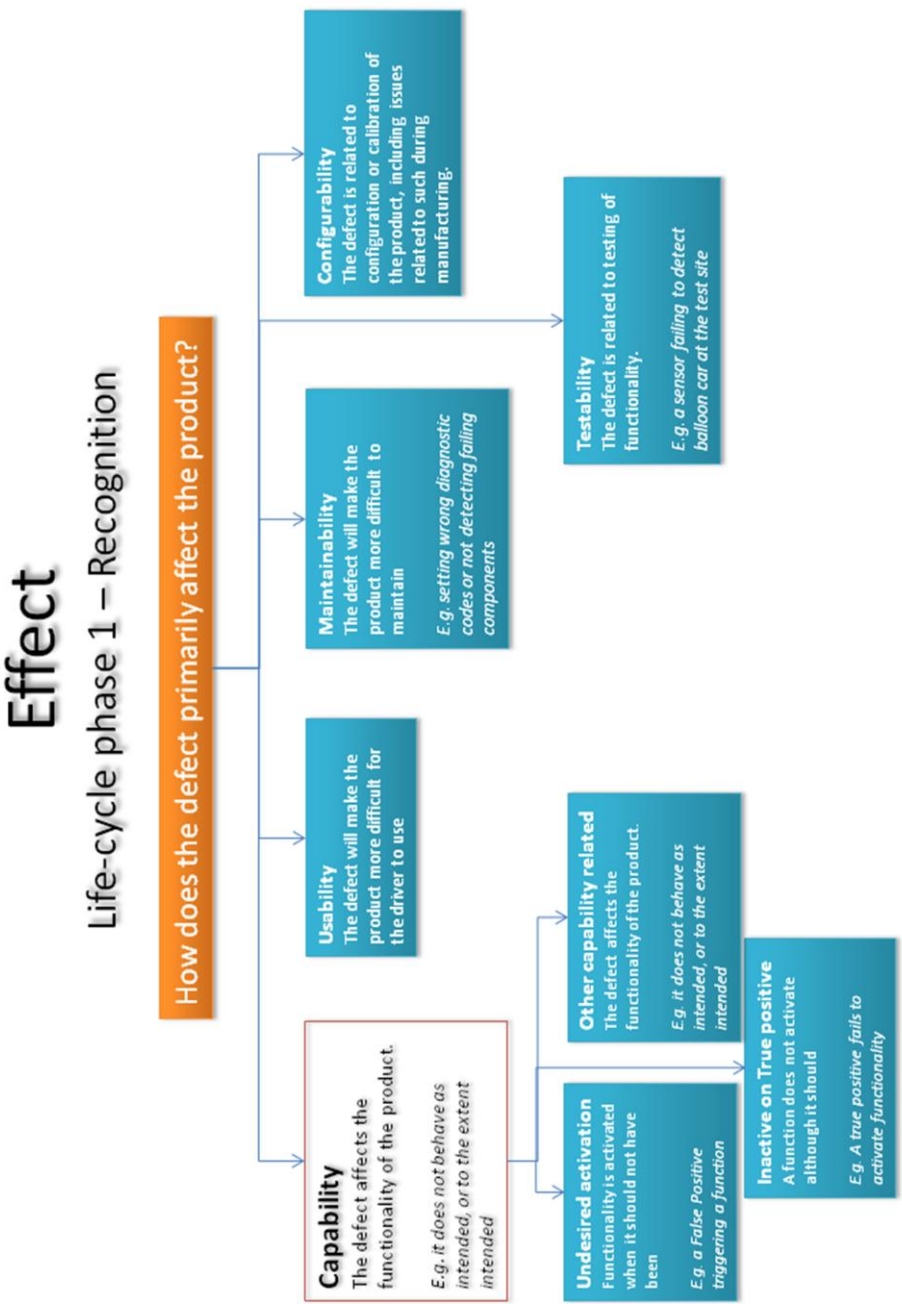

Figure 39 Classification guide for the attribute Effect

Life-cycle phase 2: Analysis – Investigating the cause of the defect

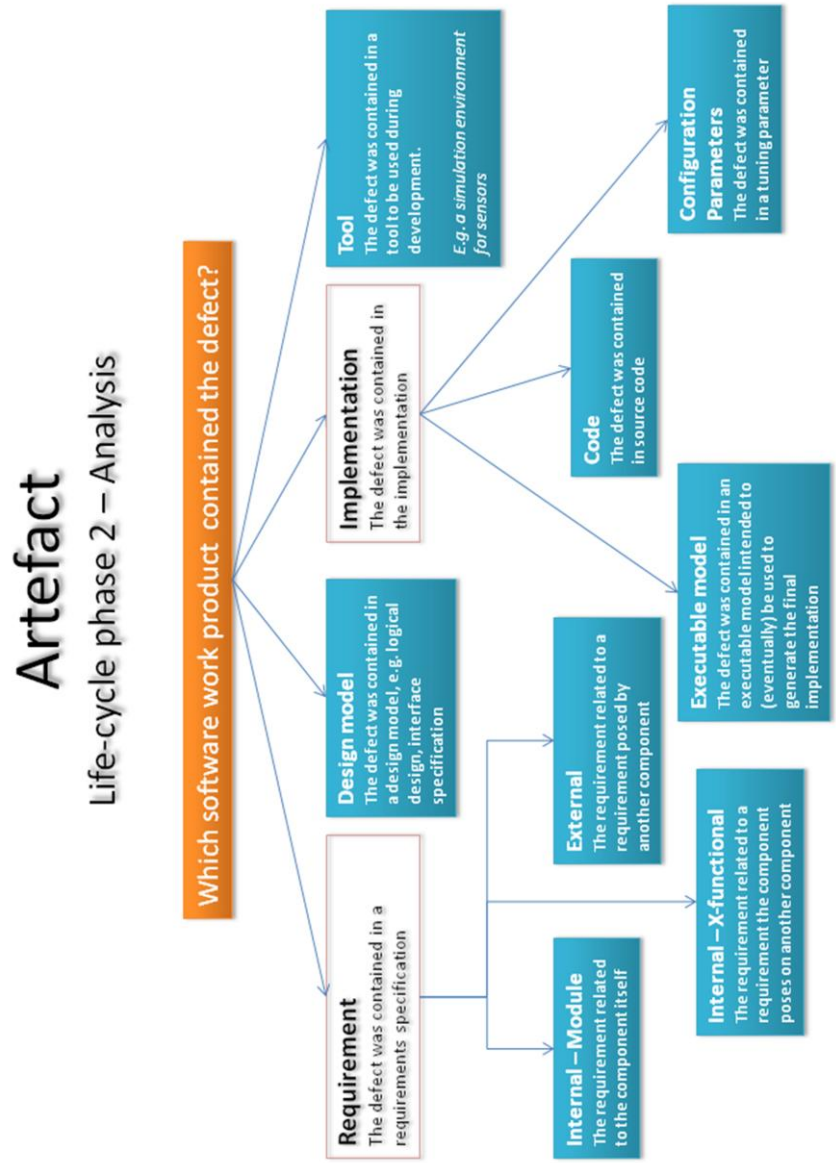

Figure 40 Classification guide for the attribute Artefact

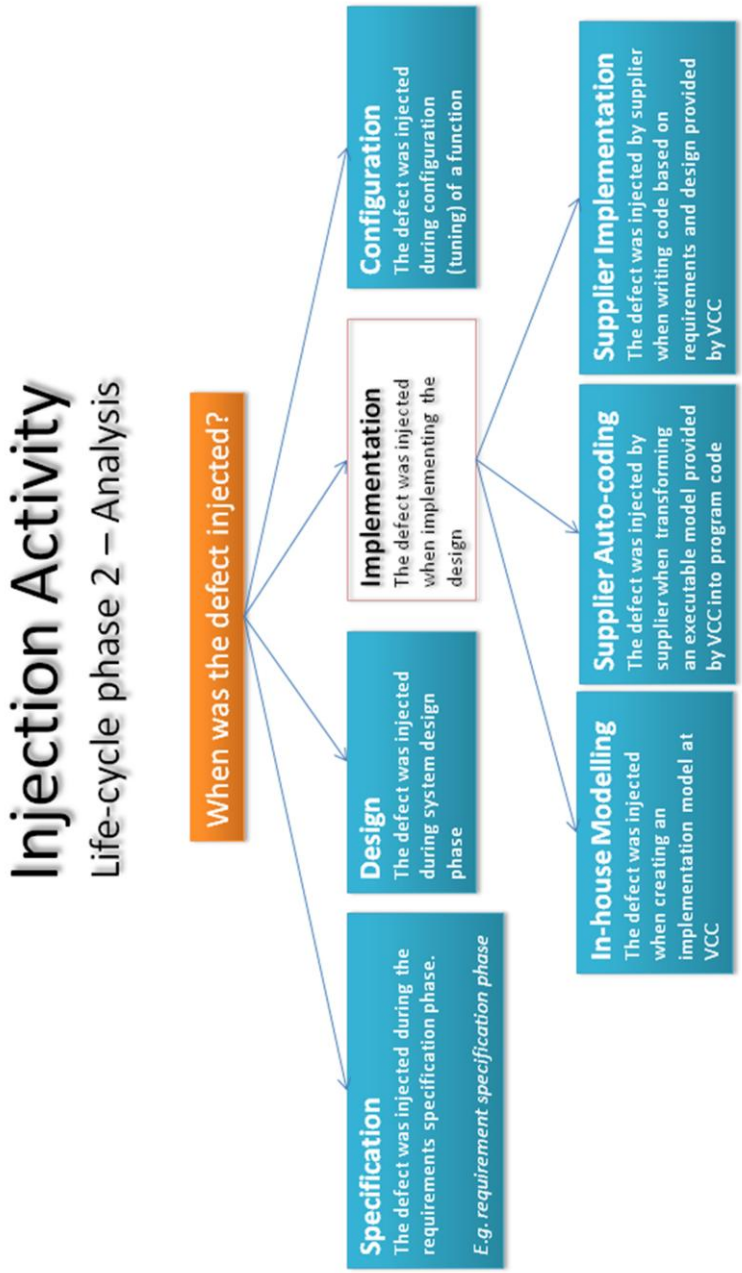

Figure 41 Classification guide for the attribute Injection activity

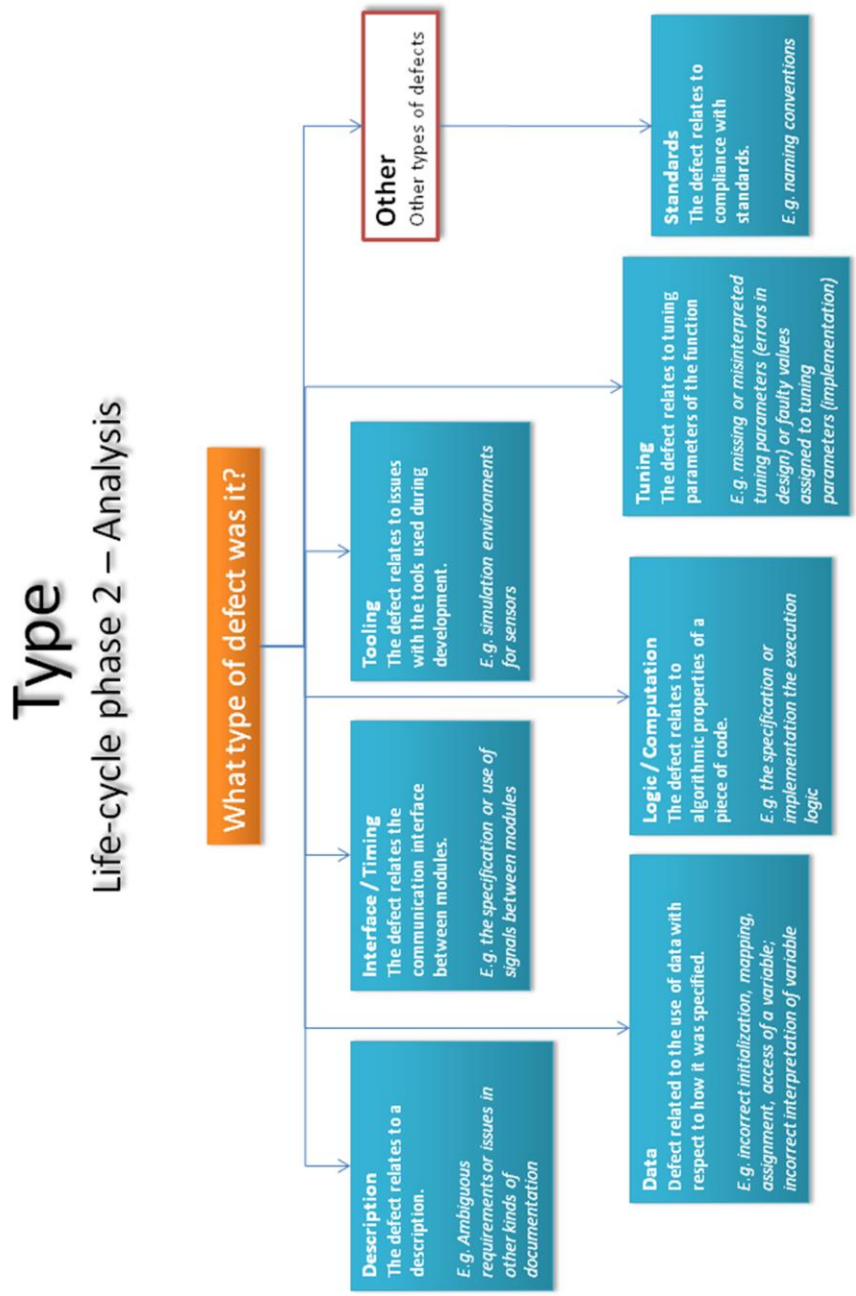

Figure 42 Classification guide for the attribute Type

Life-cycle phase 3: Resolution – The action leading to defect removal

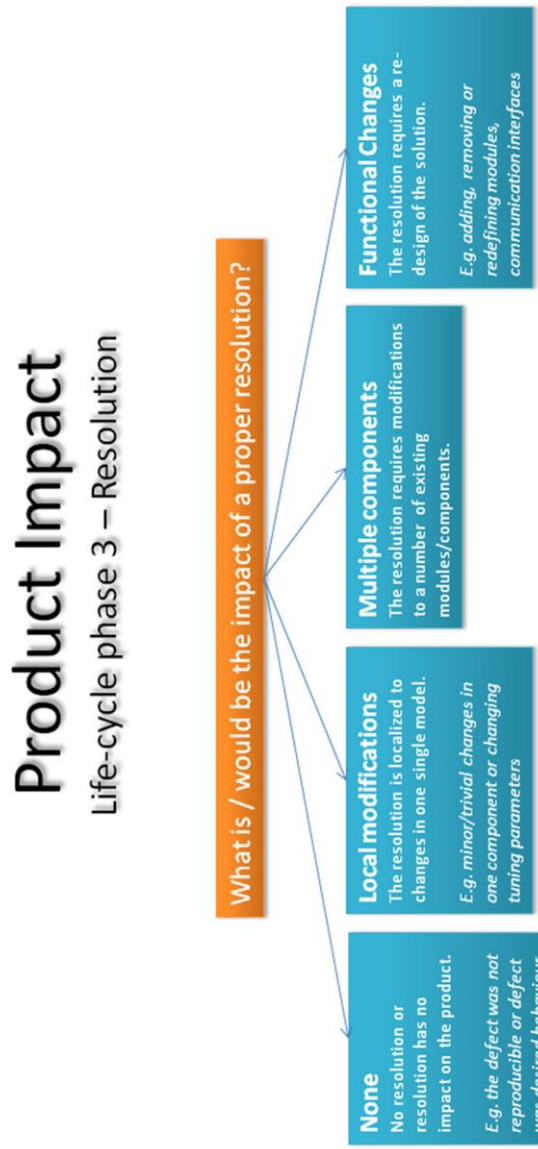

Figure 43 Classification guide for the attribute Product impact

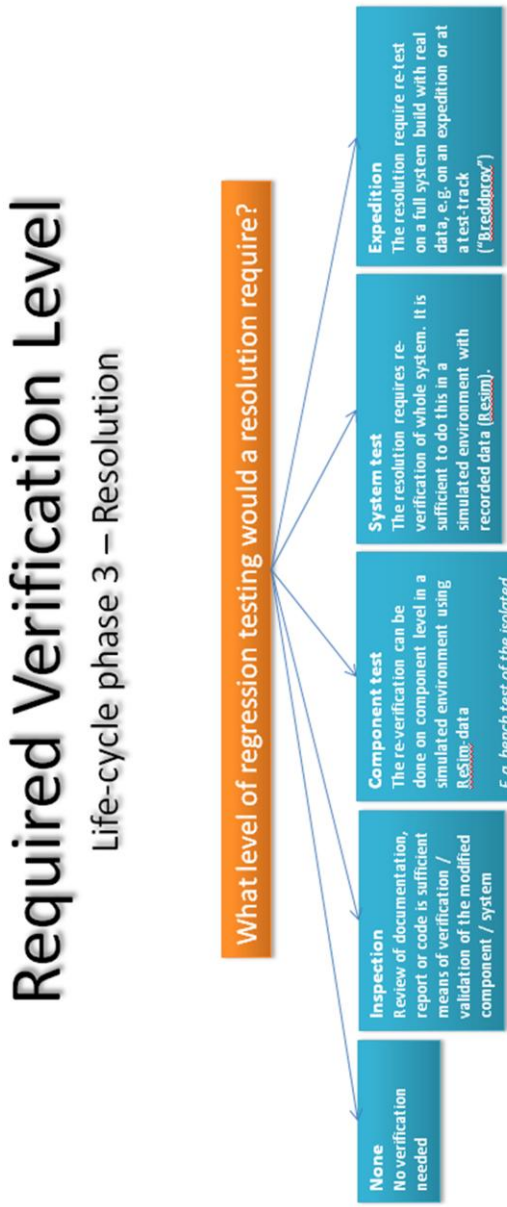

Figure 44 Classification guide for the attribute Required verification level

Life-cycle phase 4: Post-mortem – Final state of defect

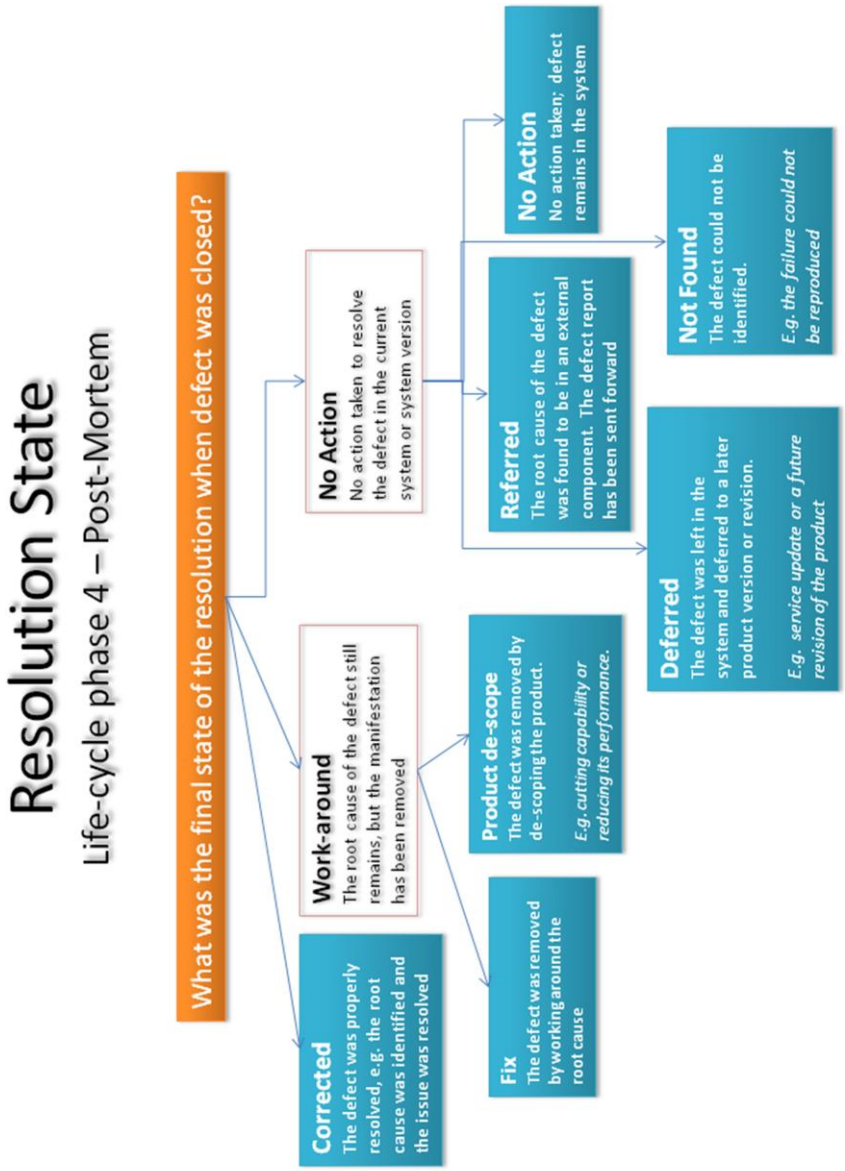

Figure 45 Classification guide for the attribute Resolution state

C

Appendix C. EXAMPLE CLASSIFICATION

In this section the classification of two example defects is described. Due to confidentiality reasons, the defects described below are construed examples. The example defects, however, are inspired by defect reports encountered during the case-study.

Example 1

The first defect caused a faulty diagnostic flag to be set indicating that part of software installed on an ECU failed. The defect was detected during testing of the vehicle at the factory manufacturing line late in the project. The problem occurred when updating the software and its configuration parameters on the ECU. In the process of deploying software and configuration parameters, the ECU was first set into a programmable mode in which a diagnostic routine is executed; it was in the diagnostic routine that the faulty error-mode was set.

During the root cause analysis it was found that the flag indicating component's (programmable) mode was stored to an incorrect output port which caused the programmable mode to be interpreted as a faulty error mode. Although the defect manifested itself during manufacturing (when software was first deployed to the ECU), it could occur during the software maintenance phase when software updates are deployed to the ECUs. It would, however, have had no effect on the normal operational mode of the component, the system or the complete vehicle (e.g. the driver would not have been affected by the defect).

Recognition phase

The date on which the defect report was submitted in the issue management system was used for the *Timing/Detection* attribute (2009-04-14). In addition, the development phase in which the detection was made was also noted (though this is redundant information, as the development phase can be derived from the date using the project plan, but it was convenient to have that information readily available). The detection time was considered (subjectively by the reporter) as significantly late in the project. The reporter considered it to be a software unit problem and that it should have been detected in an earlier test phase (either during unit testing at the supplier side, or during unit testing at the OEM side; the internal name of the preferred test phase was *U2*). The defect was, furthermore, considered to affect software, as (according to the reporter) it was probably an implementation error that led to the wrong diagnostic flag being set.

The defect was discovered during the testing of the manufacturing line when the car was assembled in factory, thus the value *Manufacturing* was selected for the *Detection Activity* attribute. As the defect in this case made it difficult to assess whether deploying the software to the ECU was successful it was considered impossible to release the software into production in its current state, and its resolution to be very urgent; the value *Immediately* was chosen for the *Urgency* attribute (as the defect was discovered late, a fix was promptly needed) and the value *Show-stopper* for the *Severity* attribute (as the software could not be released while containing the defect).

The *Effect* attribute was set to *Maintenance* as the main effect of the defect related to problems when software and configuration parameters were to be updated. The defect would, furthermore, have affected any future maintenance updates to both software and configuration parameters. Finally, the defect was considered to have no impact on any ASIL-classified requirements (if the software update had truly failed, it would have successfully triggered other diagnostic functions indicating that the component was not operating properly), it was thus considered not to have any impact on functional safety (as defined by ISO 26262).

Table 36 summarizes the classification for the Recognition phase.

Table 36. Example classification; Example 1, Recognition phase

Attribute	*Value*
Timing / Detection	2009-04-14 (Manufacturing test)
Timing / Preferred	U2 (unit testing)
Affect SW	Yes
Detection activity	Manufacturing
Urgency	Immediate
Severity	Show-Stopper
Effect	Maintenance
Functional Safety Impact	No

Analysis phase

During the root cause analysis of the failure it was found that the defect was contained in the binary code deployed on the ECU. It was, furthermore, found that the requirements clearly stated to which port the diagnostic flag should be written, and the executable model which served as base for the binary code was correctly implemented according to the requirements. Consequently, the fault was introduced during transformation from executable model to source-code; an activity done by a supplier. Therefore, the attribute *Artefact* was assigned the value *Implementation / Code*, and the attribute *Injection Activity* was assigned the value *Implementation / Supplier auto-coding*. The *Component/Asset* attribute was assigned the company's internal code identifying the software module as well as the software version.

Finally, the *Type* attribute was assigned the value *Data*, because the underlying cause of the defect related to data being written to the wrong location. Note that the *Type* attribute in IEEE Std. 1044 has a higher resolution which allows for more precision in defect analysis. In our case, however, such resolution is not possible, as the source-code (which carries the necessary information to allow for more detailed classification) is owned by the supplier. In effect, LiDeC's *Type* attribute captures a black-box alternative to the *Type* attribute in IEEE Std. 1044.

Table 37 summarizes the classification for the Analysis phase.

Table 37. Example classification; Example 1, Analysis phase

Attribute	*Value*
Artefact	Implementation / Code
Injection Activity	Implementation / Supplier auto-coding
Component / Asset	XYZ-1256
Type	Data

Resolution

The urgency of the defect (due to its late discovery) resulted in the resolution being applied in the same development phase in which it was discovered; the date (and development phase) of successful verification of the resolution was noted as the *Removal time*.

The necessary resolution was determined to be confined to a single software module, as no other components would need any modifications; thus the *Product Impact* attribute was set to *Local modification*. Finally, a test report from the supplier showing successful test of the binary component was considered sufficient means of verification; thus *Inspection* was set as value for the attribute *Required Verification Level*.

Table 38 summarizes the classification for the Resolution phase.

Table 38. Example classification; Example 1, Resolution phase

Attribute	*Value*
Timing / Removal	2009-05-09 (Manufacturing test)
Product Impact	Local modification
Required Verification Level	Inspection

Post-mortem

In the final defect life-cycle phase, the single attribute *Disposition* records what finally was done to resolve the defect. In this example, a proper resolution was applied; thus the value *Corrected* was assigned to the *Disposition* attribute.

Table 39 summarizes the classification for the Post-mortem phase.

Table 39. Example classification; Example 1, Post-Mortem phase

Attribute	*Value*
Disposition	Corrected

Example 2

The defect was found in a feature that issues an audible warning if the driver is unintentionally drifting off-lane. The function monitors the vehicle's position by tracking the lane markers in the road and gives a warning signal when a lane is about to be crossed and the driver does not use the turn signals. During the first field test (also called *expedition*) – where a mature build of the full vehicle is tested on a large variety of road types – it was found that the sensitivity of the warning was too high, resulting in frequent false alarms. The problem was detected on specific road types were the lanes were narrower than what had been anticipated.

Recognition phase

The date on which the defect report was submitted in the issue management system was used for the *Timing/Detection* attribute (2008-10-04) and noted along with the development phase in which it was detected. The detection time was considered (subjectively) by the reported as appropriate. The defect was, furthermore, considered to affect software, as (according to the reporter) it was the behaviour of the software that caused the problem.

The defect was discovered during the functional testing of the vehicle on an expedition, thus the value *Functional Test / Expedition* was selected for the *Detection Activity* attribute. As the defect in this case caused considerable nuisance to the driver on specific road types, it was considered to impossible to release the software into production with the defect remaining (the value was thus set to *Show-stopper*). As the resolution of the defect was considered to need further testing (on the problematic road type, as well as other types to ensure no regressions had been introduced) and that additional development releases had already been planned for testing, the *Urgency* attribute was set to *Next Major Release*.

The *Effect* attribute was set to *Capability / Undesired Activation* as its main effect related to false warnings. Initially, there was some confusion whether the defect should be classified as having effect on the *Usability*. However, as

the problem was not related to the *way* the user was warned (i.e. through an audible cue), the problem did not have impact on usability.

Finally, the defect was considered to have no impact on any ASIL-classified requirements, it was thus not considered to have any impact on the functional safety (as defined by ISO 26262). If, on the other hand, the feature had been able to autonomously intervene in steering or braking, it would indeed have had impact on functional safety.

Table 40 summarizes the classification for the Recognition phase.

Table 40. Example classification; Example 2, Recognition phase

Attribute	*Value*
Timing / Detection	2008-10-04 (First full vehicle functional test)
Timing / Preferred	-
Affect SW	Yes
Detection activity	Expedition
Urgency	Next major release
Severity	Show-Stopper
Effect	Capability / Undesired activation
Functional Safety Impact	No

Analysis phase

The problem occurred in running code (i.e. in the binary code of on one of the software components realizing the feature). However, as the feature was designed to be configurable (using tuning parameters) with respect to the width of the lanes, the *Artefact* attribute was set to *Implementation / Configuration Parameters*. It was, furthermore, found that the requirements specification for the feature did not take the particular road type into consideration. Additionally, the design as well as the executable model and the binary code were found to have been correctly derived and implemented from the requirements specification. Consequently, the attribute *Injection Activity* was assigned the value *Specification*. The *Component/Asset* attribute was assigned the company's internal code identifying the software module as well as the software version.

Finally, the *Type* attribute was assigned the value *Description* because the requirements did not take the particular road type into consideration.

Table 41 summarizes the classification for the Analysis phase.

Table 41. Example classification; Example 2 Analysis phase

Attribute	*Value*
Artefact	Implementation / Configuration Parameters
Injection Activity	Specification
Component / Asset	ABC-5431
Type	Description

Resolution

The resolution was applied in next major release of the software; the date (and development phase) of successful verification of the resolution was noted as the *Removal time*.

The necessary resolution was determined to be confined to one component (specifically, the configuration parameters of a software module). However, as the particular module also provided other features in the vehicle with data, and thus might be affected by the modification, the *Product Impact* attribute was set to *Multiple Components*. Finally, it was considered necessary to verify the resolution in a full vehicle build, where all features dependent on the modified software model were tested; thus *Expedition* was set as value for the attribute *Required Verification Level*.

Table 42 summarizes the classification for the Resolution phase.

Table 42. Example classification; Example 2, Resolution phase

Attribute	*Value*
Timing / Removal	2009-02-09 (Second full vehicle functional test)
Product Impact	Multiple modules
Required Verification Level	Expedition

Post-mortem

The resolution of the defect was finally done in two parts: first, the configuration parameters of the software module were modified, and; second, the requirement specification was updated with a description of the road type that caused the problem. Thus, the value *Corrected* was assigned to the

Disposition attribute. If, on the other hand, the requirements specification had not been updated, it should have been set to *Work-around / Fix* as the problem would have been mitigated, but the root cause not properly removed.

Table 43 summarizes the classification for the Post-mortem phase.

Table 43. Example classification; Example 2, Post-Mortem phase

Attribute	*Value*
Disposition	Corrected

D

Appendix D. IEEE STD. 1044 COMPLIANCE MATRIX

Table 44 shows the IEEE Std. 1044 compliance matrix (see table 3 in (IEEE, 1996)). Attributes from IEEE Std. 1044 that are not available or that are implicit in LiDeC other attributes are shown in shaded cells. Attributes that have been redefined in LiDec are marked with an asterisk(*). For more detailed information about the mapping, refer to Appendix E.

Table 44 IEEE Std 1044 compliance matrix

IEEE Std 1044-1993 attribute	*LiDeC equivalent attribute*	*Mandatory in IEEE Std. 1044-1993*	*Comment*
Actual cause	Injection activity*	√	Whereas IEEE records actual cause as the artefact that caused the defect, LiDeC records in which activity it was injected
Corrective action	*Not available*		
Customer value	*Implicit*		Implicit in LiDeC.Severity
Disposition	Resolution state	√	
Mission/safety	*Implicit*		Implicit in LiDeC.Severity

IEEE Std 1044-1993 attribute	*LiDeC equivalent attribute*	*Mandatory in IEEE Std. 1044-1993*	*Comment*
Priority	*Implicit*		Implicit in LiDeC.Urgency
Product status	Severity		
Project activity	Detection activity	√	
Project cost	Product impact*	√	The level of impact on the product – in terms of required change – is considered a better estimate of "cost" than money
Project phase	Timing / Detection	√	
Project quality / reliability	*Not available*		
Project risk	*Not available*		
Project schedule	Re-verification level*	√	Whereas IEEE.ProjectSchedule is described as "an appraisal of the amount of effort required to address the defect", LiDeC instead expresses it in terms of the amount of re-verification required. This is because verification is a costly activity, and it is a more convenient way for individual teams to estimate impact on schedule
Repeatability	*Not available*		
Resolution	Urgency*	√	Whereas the LiDeC.Urgency records how quickly the defect needs to be removed, the attribute IEEE.Resolution also records the type of resolution applied

IEEE Std 1044-1993 attribute	*LiDeC equivalent attribute*	*Mandatory in IEEE Std. 1044-1993*	*Comment*
Severity	Severity*	√	The IEEE attribute also includes whether a solution exist in the severity attribute. LiDeC, however, assesses "Severity" on the observed failure, and does thus not take the availability of a fix into consideration
Societal	Functional Safety Impact		The attribute *Functional safety impact* captures whether the defect may risk causing harm to persons (as defined by the ISO 26262 (ISO/DIS, 2011)). This maps to the IEEE Std. 1044 attribute *Societal* in that it captures data about the impact of the defect on the environment (e.g. driver, passenger or other persons in the vehicle's surroundings).
Source	Artefact	√	
Suspected cause	*Not available*		
Symptom	Effect	√	
Type	Type	√	

E

Appendix E. MAPPING BETWEEN IEEE STD. 1044 AND LIDEC

This appendix provides a detailed mapping between the attributes in LiDeC and in IEEE Std. 1044.

Table 45 Mapping between IEEE Std. 1044 and LiDeC

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
Recognition	**Project Activity**	What were you doing when the anomaly occurred?	**Detection activity**	How was the defect detected?	
	Project Phase	In which life cycle phase is the product?	**Timing / detection**	When was the defect detected?	
	Suspected cause	What do you think might be the cause?	*n/a*		Speculation of the cause would mainly be of interest when analysing the fault. This is done as part of the defect management process at the company, but is not of interest for our analysis
	Repeatability	Could you make the anomaly happen more than once?	*n/a*		This attribute is captured by Disposition
	Symptom	How did the anomaly manifest itself?	**Effect**	What requirements category does the defect affect?	The IEEE Std. 1044 has a very detailed symptom classification. In our approach we analyse instead the type of impact the symptom would have on the product; e.g. Capability, maintainability etc
	Product Status	What is the usability of the product with no changes?	**Severity**	How severely does the defect affect the product?	

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
	n/a		**Timing / Preferred**	When should the defect have been detected?	The attribute records the developers (subjective) opinion on whether this defect's discovery was timely or if there was an earlier project phase in which it reasonably should have been detected. No such attribute exist in IEEE 1044
	n/a		**Affects Software**	Does the defect affect software?	As the development of automotive software is a hybrid of hardware and software development, and that our main interest lies in studying aspects related to software development, we use this attribute to make an initial coarse filtering of the defects
Investigation	**Actual cause**	What caused the anomaly to occur?	**Injection Activity**	When was the defect introduced in the product?	Closely related to IEEE.Source and LiDeC.Component. Whereas the IEEE maps this on product parts LiDeC captures the activity in which the defect was injected; i.e. a defect discovered in code may have been introduced due to ambiguous requirements.
	Source	Where was the origin of the anomaly?	**Artefact**	Which software work product contained the defect?	

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
	Type	What type of anomaly/enhancement at the code level?	**Type**	What type of defect was it?	Directly mappable, though LiDeC use a much higher abstraction level of the selection of types. There were still cases where the distinction between types was not straight-forward – mainly because the types were not easily understandable (rather than lack of understanding of the defect itself)
	Action supporting data item		**Component/ Asset**	Which design component contained the defect?	The attribute captures which part of the product contained the defect. This relates to IEEE.ActualCause and is also part of the supporting data items in the Action life-cycle phase, although that data item captures which part of the product will need changing (which may not be the same as the one containing the defect!)

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
Action	**Resolution**	What action to take to resolve the anomaly?	**Urgency**	How urgent is it to resolve the defect?	LiDeC.Urgency also maps to the IEEE.Priority attribute. However, investigating the defect to arrive at the priority requires resources; we have in LiDeC chosen to record the Urgency attribute on failure-level instead of on fault level. Consequently, 'Urgency' relates to how urgent it is to remove the manifestation of the fault rather than the fault itself (which the IEEE.Priority attribute specifies)
	Corrective action	What to do to prevent the anomaly from happening again	*n/a*		Whereas IEEE records the exact resolution we have chosen instead to record the extent of impact the resolution would have on the product (see IEEE.ProjectCost).
	Action supporting data item		**Removal Time**	When was the defect closed?	LiDeC captures the time of closing the problem report (regardless of the state of the resolution) in order to be able to measure the longevity of defects and the project workload. This information is interesting as it serves as a measurement of the pressure on the project – assuming mistakes are more likely to be made under pressure one would like to keep the number of open defects to a minimum

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
Impact Identification	**Severity**	How bad was the anomaly in more objective engineering terms?	**Severity**	What would the impact on the product be if defect remain in system on release?	Also see IEEE.ProductStatus
	Priority	Rank the importance of resolving the anomaly (subjective)	**(Urgency)**	How urgent is it to resolve the defect?	See the IEEE.Resolution attribute
	Customer value	How important a fix is to customers?	*n/a*		This is implicit in the LiDeC.Severity attribute
	Mission / Safety	How bad was the anomaly with respect to project objectives or human well-being?	*n/a*		This is implicit in the LiDeC.Severity attribute
	Project schedule	Relative effect on the product schedule to fix	**Required Verification Level**	What level of regression testing would a proper resolution require?	Required effort to apply a resolution is not only captured by the amount of necessary modification to the product. As automotive software have very high reliability requirements, V&V activities require substantial amount of resources. This attribute records the estimated level of regression testing that a proper resolution would require (as order of magnitude)

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
	Project cost	Relative effect on the project budget to fix	**Product Impact**	What would the impact on the product be if a proper resolution was applied? Value is intended as order of magnitude – from no impact, local modification to a system re-design	Whereas IEEE.ProjectCost specifies to record an appraisal of the cost of a resolution in dollars, LiDeC instead records an estimation of the impact a resolution would have on the product (in terms of the amount of modification needed). We stipulate that the impact of a resolution on the product will correlate with the cost of applying it; the impact, however, is easier to estimate by the person reporting the defect
	Project risk	Risk associated with implementing a fix	*n/a*		
	Project quality / reliability	Impact to the product quality or reliability to make the fix	*n/a*		

Life-Cycle Phase	IEEE Std. 1044	Description	LiDeC	Description	Mapping comment
	Societal	Impact to society of implementing the fix	**Functional Safety Impact**	Does the defect have an impact on a software component with ASIL-classified requirements (ISO 26262)?	The attribute Functional safety impact captures whether the defect may risk causing harm to persons (as defined by the ISO 26262 (ISO/DIS, 2011)). This maps to the IEEE Std. 1044 attribute Societal in that it captures data about the impact of the defect on environment (e.g. driver, passenger or other persons in the vehicle's surroundings). Note, the Functional Safety Impact attribute is captured in the recognition phase
Disposition	**Disposition**	What actually happened to close the anomaly	**Resolution State**	What was the final state of the resolution when defect was closed?	Directly mappable (values modified)