UPPSALA
UNIVERSITET

UPTEC STS 18010
Examensarbete 30 hp
Juni 2018

Evaluation of code generation in agile software development of embedded systems

Laura D'Angelo

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluation of code generation in agile software development of embedded systems

Laura D'Angelo

Generating code from software models is considered to be a new generation leap within software development methods. The objective of this M.Sc. project is to evaluate how different approaches to modelling and code generation affect embedded systems software development and propose recommendations on how to improve software development. Two product areas at Saab Surveillance EW Systems in Järfälla, Stockholm, are used as study objects.

A research overview is made to highlight themes regarding modelling, code generation and software development in general. Based on these, interviews are held with system engineers and software developers at each product area, where they use different modelling and code generation approaches. The two development processes are described thoroughly. Challenges and advantages related to each area's approach are investigated.

Software development within product area A is affected by the product complexity and the larger scale of the development, including projects running over a longer time with more teams involved. Recommendations include enabling code generation by aligning it with other investments on process improvement and limiting the approach to generating some system components. Software developers within product area B can use full code generation, enabled by the limited product complexity. The product area is affected by software standards and external requirements on the process. Recommendations include extending the modelling approach to make it easier to trace functionality from system to software level. Conclusions are that both product areas can apply modelling and code generation to more software development activities to improve their respective development processes.

Handledare: Viktor Edman
Ämnesgranskare: Bengt Jonsson
Examinator: Elísabet Andrésdóttir
ISSN: 1650-8319, UPTEC STS 18010

Populärvetenskaplig sammanfattning

En av de metoder som sprider sig inom industri och utveckling av inbyggda system är modellering. Modellering inom mjukvaruutveckling innebär precis som det låter, att använda visuella och logiska element för att planera, designa och implementera mjukvara. Modellerna kan användas för att diskutera lösningar och rita bilder, men också för att generera exekverbar kod. Att börja modellera ett helt system eller delar av mjukvara och generera kod från modeller kan göras på oändligt många sätt. Antalet språk, verktyg, arbetssätt och metoder för kodgenerering är många. För en organisation är det viktigt att förstå hur, var och när ett koncept bör appliceras och användas för att det ska lyckas och göra mjukvaruutvecklingen konkurrenskraftig. Detta kräver en förståelse för både valda metoder och tekniker, men också interna strukturer, processer och arbetssätt.

Genom en kvalitativ undersökning studeras två produktområden på Saab Surveillance i Järfälla utifrån hur de använder modellering och generering av kod i utvecklingen av olika typer av system. De arbetar med olika sorters produkter och under skilda förutsättningar, där de har olika metoder för utvecklingen av inbyggda system. Semistrukturerade intervjuer har genomförts med systemingenjörer och mjukvaruingenjörer i båda produktområdena för att utreda svårigheter och positiva effekter med de valda metoderna. Utifrån resultatet utreds hur olika kontexter kräver olika typer av metoder och hur det går att anpassa en metod till förutsättningarna i en mjukvaruorganisation.

Produktens komplexitet påverkar tydligt produktområdenas val av metoder, där komplexa produkter möter mer svårigheter i att implementera kodgenerering av all mjukvara. För mer komplexa produkter med fler gränssnitt bör kodgenerering därför vara avgränsat till vissa systemkomponenter, medan för produkter med en enklare produktstruktur kan kodgenerering göras för all mjukvara. En avgörande faktor för lyckade processer oavsett produktområde visar sig vara att väva samman systemingenjörers arbete med mjukvaruingenjörers arbete med hjälp av modeller. Det gynnar kommunikation inom och mellan team, samt ger fördelar relaterat till flera av de utmaningar som mjukvaruutveckling medför.

Studien kan användas för att besluta kring utvecklingsmetoder och ta strategiska beslut kring hur modellering och kodgenerering bör användas i en process. Det kan också användas för situationsmedvetenhet inom mjukvaruutveckling av inbyggda system.

Acknowledgements

This thesis has during spring 2018 been conducted as a final part of the Master Programme in Sociotechnical Systems Engineering at Uppsala University. The report is a result of a research project at Saab Surveillance EW Systems in Järfälla, Stockholm.

I wish to address my uppermost gratitude to my mentor Viktor Edman at Saab for his patience and commitment during the whole process. I want to thank my supervisor Bengt Jonsson from the Department of Information Technology at Uppsala University for contributing with valuable insights.

To all Saab employees whom I encountered during this thesis, whether an interviewee or simply engaged in my work, and to the department that trusted me with this task – thank you.

Last but not least, this spring wouldn't have been the same without sharing a writing room at Saab with Sofia J, Fanny, Felicia, Araxi and Sofia B. I'm so happy for our new friendship and for you always giving me reasons to laugh during the toughest days.

Laura D'Angelo

Uppsala, May 2018

Table of Contents

1. Introduction

The use of modelling methods in industry intensifies. The functionality and design of a system can be modelled at the beginning of the development process. Models are in some way used to implement the system, hardware or software, where models are used to make important artifacts like test cases or documentation, or more detailed executable models (Brambilla et al., 2017). Liebel et al. (2014) state that among companies in the embedded systems domain, models are mainly used for documentation.

Model-driven development is a development process where models are used as the primary artifact of the development process. The idea of model-driven development is that the gap that sometimes exists between the model of a system (an abstraction on any level) and the implementation (the real system) can be reduced or dissolved (Jörges, 2013). In the model-driven development approach, an essential process for software development is that models are used to generate code for implementation of the system (Brambilla et al., 2017).

Code generation uses a model to generate executable code. The idea is similar to a compiler that creates executable binary files from source code, but model-driven development is to generate source code from models (Brambilla et al., 2017). Applying code generation techniques thus raises the level of abstraction at which developers usually work, because some of the logic generally explained in code is represented by even more abstract elements like boxes, arrows, and different symbols.

A fundamental question for the whole modelling and code generation paradigm is which costs, benefits, and challenges it brings to an organization and the development and maintenance of software (Chaudron, 2017). Each organization is unique, and there can't be a universal method for using modelling and code generation. There is a need for understanding the organization's unique context and requirements to know what approach will make the software development as competitive as possible (Brambilla et al., 2017).

The success of implementing a model-driven approach relies on a careful introduction of the method into the daily practice and the existing technological and social context (Selic, 2003). Since software is developed by humans and processes are built for human work, the model-driven process may reform the software development in terms of new roles, new skills and new patterns of work (Brambilla et al., 2017). The understanding of its effects needs to have both an organizational and a technical perspective (Trendowicz, 2013). In general, the introduction of new software development methods is related to risks of failure if the implementation is not done cautiously (Anda et al., 2006). All software development organizations need to carefully consider the fit of a new or reinvented process.

1.1 Problem Statement

Electronic Warfare (EW) Systems is a business unit within Saab Surveillance. It provides services for protection, detection, location, and monitoring utilizing the electromagnetic spectrum. The unit strives for developing solutions that are easy to integrate, operate and support and at the same time are high performing. The development departments within EW business unit are responsible for software and hardware development of EW products. They develop high performance, large-scale software and hardware for embedded systems.

The use of models as development artifacts has been deployed widely in the organization and the development of next-generation EW products. However, the current development processes at EW Systems vary in methodologies, tools, and frameworks between projects. Models are used in projects related to many products, but the modelling approaches differ fundamentally.

In the development process within product area A at EW Systems, the functionality of a system is modelled by system engineers. At this stage, a system is modelled with its high-level functionality for hardware and software. After systems engineers have created an overview of the system, the system models are used for generation of documents. Software developers in projects related to product area A use the documentation to manually translate descriptions of system components into code.

Within product area B projects at EW Systems, the methodology is another. The functionality of a system is not planned with the help of models. It is planned with extensive documentation, which is later on delivered to software developers. Software developers interpret the documentation and produce software models. Software models are used as a central artifact for all software development tasks, including full code generation.

The differences in development methods at the same company are considerable. For all companies, teams, and actors working with embedded systems development, it is crucial to effectively and dynamically keep up with changes in the industry of software development. Changes can mean adaption of innovative technologies in their products but also internal process management. Keeping up with standards in the industry does not mean incautiously applying new methods, but making choices supported by literature studies and experiences.

This thesis intends to investigate how models are used in industry and the context of agile embedded system and software development. Evaluating advantages and disadvantages of different approaches for different products is a clue to understanding how to make a modelling and code generation approach successful. The thesis can provide insights on different contexts of embedded systems software development affect modelling and code generation approaches.

1.2 Research Questions

This study aims at understanding how model-driven development is used within the embedded software development domain. Three research questions are formulated to support the objective of the study. The research questions are answered with help from a case study performed in two separate product areas within the organization at Saab Surveillance that uses entirely different approaches to modelling and code generation.

RQ1: How is modelling and code generation used in the software development processes within product area A and product area B at Saab?

RQ2: What are the difficulties and advantages of different uses of modelling and code generation approaches within software development of embedded systems?

RQ3: How do different contexts of embedded systems software development affect modelling and code generation approaches?

The material found can be used for situation awareness, strategic decisions and effective implementation, and is meant to cover a broad socio-technical view of model-driven approaches with code generation. Furthermore, findings can be used for groups and organizations in the embedded systems domain concerned with changing, implementing or evaluating model-driven approaches.

2. Method

This section covers the overarching method for answering the research questions. A research overview is conducted and evaluated, followed by a case study at Saab Surveillance EW Systems in Järfälla, Stockholm. This chapter will cover the idea of the approach.

2.1 Research Overview Motivation

The research overview helps creating themes on modelling and code generation that can be used as topics in interviews. It also supports the creation of themes that concern the success of software development. The themes are meant to be relevant in time and up-to-date with research in the field as well as somehow related to process changes and internal work flow. Findings are used to formulate interview questions when investigating modelling, code generation and software development at Saab.

The research overview is written with support from databases and libraries. Key words are used for initial searches and only peer-reviewed material is studied. When a relevant article, book, report etc. is found, references from that piece is used for further research (Ejvegård, 2009). By following references deeper into the specific area of concern, it is possible to cover the most central literature for the chosen theme. The relevance of a source is evaluated primary by reading abstract, keywords or a summary, and secondary by looking at the table of contents, headings and through a quick skim of the content. If a piece of work is considered relevant, it is read carefully and learnings from it are summarized and categorized (Ejvegård, 2009). When all data has been summarized and categorized, data concerning a specific theme in code generation or modelling is used to compose an extensive walk-through of that theme from different sources of literature.

2.2 Case Study Motivation

Using a case study object can help digging deep into a problem with help from specific activities, people and processes that are real. The understanding of findings in a case study is easy because of the actual and detailed descriptions (Eriksson and Wiedersheim-Paul, 2011). With help from a deep description of a setting, it is easier to generalize and at the same time positioning findings in the research field. Conclusions then have to be viewed as indications of what reality looks like (Ejvegård, 2009). Eriksson and Wiedersheim-Paul (2011) describes that case studies are good for illustrating areas where some aspects are unknown to create hypotheses or as a method to illustrate changes in an organization through deep understanding of the specific context.

2.2.1 Process mapping

The idea of process mapping is to map the process of doing something and identifying value-adding and non-value adding activities and steps (Ali et al., 2016). In normal process mapping methods, like value stream mapping, there is usually a focus on

resources and material. In software development, the analysis of material flow will focus on artifacts related to software development. Artifacts can mean requirements, diagrams and models. For software development it will be appropriate to also capture documented or verbal information, formal or informal communication and the competence or key person carrying out value-adding activities (Ali et al., 2016).

Process mapping is a support in this thesis for gathering information about the development process in the case study. By using process mapping, the interviewer can get an understanding of the whole system and software development process. This is essential for evaluating how modelling and code generation affect the work. The process will be described primarily in textual form, but with some visualization if necessary. Mapping previously mentioned artifacts, information, communication and activities can give insights like bottlenecks, challenges, delays and unnecessary work.

2.3 Reliability

Regarding reliability, the context of the case study is important to consider when generalizing its findings. The chosen method should give a result independent of authors, and depending on what degree of generalization wanted – also independent of the investigated units (Eriksson and Wiedersheim-Paul, 2011). The author is responsible for building the measuring instrument (the investigation method), which implies a risk that the quality of the instrument is affected. This is the result of human factors and human impact, which may be the result of ignorant choices (Ejvegård, 2009). The support from the literature study helps the author to target relevant themes that are commonly mentioned in modelling and code generation literature, and apply it to the case study.

The case study has been done in a large company that develops software for embedded large-scale systems in the defense industry. The software organization has different departments for different tasks and fixed roles for employees in these departments. The results may be applicable to organizations working under similar contexts but not to smaller organizations with more integrated departments and variable roles. Important factors to consider regarding to reliability will be the development domain (embedded, real-time), method (agile/Scrum), project magnitude (large-scale, long-time) and the existing level of adoption of model-driven or model-based methods at the organization of investigation.

Reliability demands that in another setting, with another author but under similar contexts, the results would be the same (Eriksson and Wiedersheim-Paul, 2011). When results are not measurable with numbers, one must be aware of that all results are interpreted by the author (Eriksson and Wiedersheim-Paul, 2011). Ali et al. (2016) mentions that there can never be “one size fits all” regarding to software methodology, approach, tool, technique or standard. This is important to consider when making sense of the results and conclusions.

3. Research Overview

3.1 Modelling

Depending on industry, product or service, the purpose of modelling and who does it can be different. Within a company, the purpose of modelling as well as stakeholders and tools can vary between departments and units (Heldal et al., 2016). Modelling allows investigation, verification, documentation and discussion about properties of systems before they are put into production (Brambilla et al., 2017).

Software modelling and systems modelling are fundamentally different. System models explain the whole embedded systems domain in terms of functionality, they describe the behavior of the whole system at an abstraction level to the system as a whole. Software models on the other hand can include behavioral specification at an executable level. This means that it is not possible to use the same kind of description for the system as a whole, since the abstraction levels will differ between system and software. This leads to a number of models on different levels that describe a complex system, where not all models are compatible to each other (Bassi et al., 2011).

In the literature on modelling there are many different names for the use of models as an artifact in the development process. The most common method names are versions of "model-driven" and "model-based" methods. Model-driven methods (model-driven systems engineering and model-driven software development) uses models as the primary artifact for the development process. This means that the models are used throughout the process, from planning to implementation. A key activity in model-driven software development is that programs are generated from their corresponding models, called code generation (Selic, 2003). Model-based methods (model-based systems engineering or model-based software development) is the milder version of modelling methods. This means that models do not play a central role during the whole development process, instead they serve as some kind of support, blueprint or visualization for groups during development. This can mean that models are made as a support for understanding how to fulfill the system requirements and abstractly plan the development. Here, models only serve as some kind of planning help, sketches or blueprints (Brambilla et al., 2017).

3.1.1 System modelling for systems engineering

Systems engineering is described by the International Council on Systems Engineering (INCOSE, 2011):

"Systems engineering is a discipline that concentrates on the design and application of the whole (system) as distinct from the parts. It involves looking at a problem in its entirety, taking into account all the facets and all the variables and relating the social to the technical aspect."

Grösser et al. (2017) describes how model-based approaches in systems engineering is the solution to handling technical management of complex systems efficiently. It is an approach which is widely used in the system modelling domain. System modelling is complex and unpredictable, which results in that it can't be formalized or automated completely. What elements are needed in a model is unknown, and modelers go through an extensive learning process and try to understand how the system works. Every problem is different and leaves the modeler with various decisions to take.

In the same domain, new systems will have common elements with older systems. Therefore, it is important to establish some kind of reuse of knowledge, experience and design decisions (Marincic et al., 2013). It is described how a common problem in modelling is that when a model is delivered or used, the knowledge is lost or a transfer of learnings doesn't take place.

3.1.2 Software modelling for software development

The dependency on technology in all of society has become almost equivalent to the dependency on software, which is continuously increasing (Trendowicz, 2013). The also increasing complexity of software systems brings challenges for software engineers as they need to deal with a variety of development-related artifacts such as changing processes, languages, frameworks and platforms (Jörges, 2013). To handle these challenges, many organizations have set up formal processes for software development, in which activities for development are standardized (MacCormack and Verganti, 2003). As software development is getting more complex and more important, there is an urgent need for all organizations to strive for reducing development costs and time to market. In growing concurrence, the best functionality and quality is also a must for software competitors (Trendowicz, 2013).

Software development is a process which is depending on developers and the competence and capabilities of those. Selic (2003) describes how early-generation technologies in software development has affected the field. A great amount of intellectual effort and financial means has been used to maintain and upgrade endless lines of code. Because of great investments in time and effort, both individuals and organizations can be unapproachable with fundamental changes in programming techniques. Instead, process improvement can be more culturally acceptable and easy-adopted in software organizations (Selic, 2003).

Since the introduction of the compiler, model-driven development is said to be the first real generation leap in software development (Selic, 2003). The key is in improving the whole software development process and re-thinking the meaning of software culture and artifacts. This is also why it is mentioned that there could be resistance among developers to moving more towards model-driven methods (Whittle et al., 2014). Automating traditional and well-established tasks as well as "constraining creativity" in manual tasks are issues not to forget. Model-driven methods are described by Whittle et al. (2014) to

put even more control into the hands of architects, whose design decisions are even more emphasized in the development process.

3.1.3 Modelling languages

Modelling languages are a main ingredient of all modelling approaches. A modelling language has specific graphical and/or textual representations, in which designers or architects can model a system in a consistent way. One of these modelling languages is the OMG Systems Modelling Language (called SysML from now on). This language is designed for modelling of different and complex systems engineering problems (Object Management Group, 2017). It can be used as a language to model components that surround software. This means effective support for systems engineers in terms of creating and modeling requirements, specifications, analysis, verification, validation, behavior, structure, allocations and constraints (Object Management Group, 2017).

While SysML is a modelling language for systems, UML is a modelling language for software design and implementation. SysML is compatible with most UML models and can be used to establish relationships between different models. SysML and UML can be used with any drawing tool that supports it (Bassi et al., 2011). This thesis is meant to be as tool/vendor-neutral as possible but will refer to SysML and UML as they are the bases for modelling languages used at the product areas represented in the case study.

3.2 Code Generation

3.2.1 What is code generation?

Code generation is described by Rumpe (2017) as "the automated transition from a model to an executable program". Code generation is a cornerstone of the model-driven software approaches. Code generation uses higher level models to generate code, where target languages are often high-level languages such as Java or C++ (Jörges, 2013). Code generators are to models what compilers are to source code, which can be seen in Figure 1. Code generators can be called model compilers since they compile the model down to a lower level language (Brambilla et al., 2017).

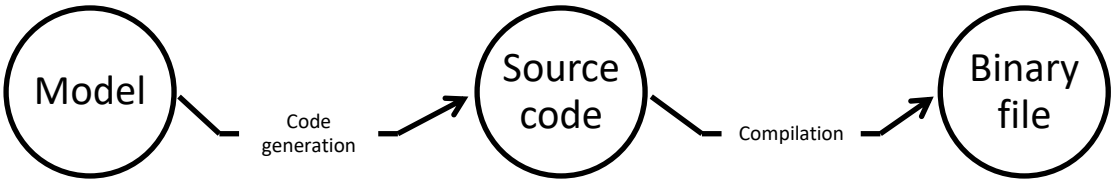

Figure 1. Code generation understood as a compiler.

The task of automatically deriving an implementation of a system from a model is the essence of code generators (Jörges, 2013). This can be done by a rule-based template engine which means that the text generated is based on specific rules. When source code

is generated, an IDE tool can be used to process the produced code if needed (Brambilla et al., 2017).

3.2.2 Advantages and disadvantages

For implementing a code generation method, there are choices to make and means to be aware of. Several advantages and disadvantages with different aspects of code generation are discussed in literature.

There are several advantages that are mentioned for code generation. The overall bridging of the gap between models and implementation brings an obvious and interesting change to software development processes. Jörges (2013) mentions that a code generator consistently does fixes and modifications to all corresponding parts of the code automatically. Brambilla et al. (2017) which describes it as less bugs and errors in the generated code than it would have been accidentally in hand-written code by developers. Zhang and Patel (2011) mentions similar matters as decreasing intensity of defects, and chooses to call it the quality of the code.

Also related to the developer as a human being is the standard of the code. Generated code is produced in a standard programming language without the style of a specific person. This could make it more consistent and easy to understand. In contrast, a drawback could be unfamiliarity for developers with this new, standardized style (Brambilla et al., 2017). Even if it behaves just like they would expect it to, it could be hard to accept how the code looks, or even hard to understand it.

Zhang and Patel (2011) describes a system where 93 % of the software was generated from models. The project was agile and in the end, they could show that the number of produced source code lines per developer increased threefold compared to manual code writing. On the contrary, Whittle et al. (2014) underline that productivity improvements can make an organization blind to drawbacks that may occur. Gains in one aspect in one branch of a company may be outweighed elsewhere. Substantially increasing productivity can’t be expected at the same time code generation is implemented (Brambilla et al., 2017). The process requires a step-by-step integration and to be adapted and accepted in the organization. Data from a study by Whittle et al. (2014) suggests that some of the proven productivity wins of implementing a model-driven approach will be outweighed by the increased training time and costs as well as organizational changes. Significant training is needed for the use of a model-driven method, because of the need to learn modelling languages, modelling tools and modelling methods (Liebel et al., 2014).

Other advantages discussed is that code generation would protect the intellectual property of the modeler, which can be a systems engineer or a software developer (Brambilla et al., 2017). Disadvantages discussed is that the average model creation effort when code is to be generated from the models is much worse than when models are not used for code generation (Chaudron et al., 2012). When models become large, they can also become

too complex. Liebel et al. (2014) also mentions that available tools are too immature for the large-scale adoption of model-driven methods.

For software development of systems that are safety-critical (e.g. in avionics) it is essential that the automated translation in a code generator leads to the desired results in terms of quality (Jörges, 2013). It is important that quality measures of a code generator can be assured, through validation and verification of its trustworthiness. There can be systems or software standards, like industry standards, which must be fulfilled in the development of a system, and the chosen approach must fulfill these.

3.2.3 Abstraction level

The abstraction level describes the level of detail at which a system can be viewed. If the amount of complexity is low, the system is viewed from a high level. If the system is viewed on a low level, the amount of detail is high. The highest level a system can be viewed is as one black box. Opening the black box and finding a handful of new boxes would be going into a lower level. With no boxes left (if possible) the abstraction level is the lowest. Systems at a high level of abstraction are often used for communication and visualizations for groups rather than implementation or code generation (Chaudron et al., 2012).

Abstraction levels are often used for shielding software developers from too much details when it isn't relevant (Jörges, 2013). Not only shielding of complexity can be helpful, but also making problems easier to understand and solve by choosing the abstraction level which suits what subset of a problem best (Bassi et al., 2011). Working with abstraction levels is not new in the field of software development, with obvious examples from the introduction of compilers (Jörges, 2013). The level of abstraction was raised to shield developers from machine code by using high-level languages instead. For designers, there is an interest in only viewing the information that is meaningful at a given task. Many views of a system are needed and the description of a system is often a collection of models. These can be at different abstraction levels and of different subsystems (Bassi et al., 2011).

Models on a high level of abstraction can be reused easier than code (Labrosse, 2007). Artifacts could be used as templates and reused for future software development (Brambilla et al., 2017). It is easier to change core concepts at a higher level of abstraction than at a lower. At this level, not all specifications have been decided (Labrosse, 2007). The model can be redeployed into other software platforms or implementation environments. At the same time, making models that match a variety of needs in a variety of projects, a generalized one, could be hard. It is discussed by Glass (2003) to be most applicable to a product line where projects attack similar problems in the same application domain.

In large organizations, different tasks in the software development process are often made by separate groups. When labor is divided, it is important to consider all handovers (Glass,

2003). These can occur when changing abstraction level, from requirements to design or design to code. The art of handovers is something that code generation cannot avoid being related to since it is the automatic translation between abstraction levels, sometimes covering the gap of a handover.

For code generation approaches, a software developer would use models to build programs. When using graphical notations instead of textual, there are cognitive gains. Visual programming languages have proven this and support the use of graphical notation as friendlier to the nature of how the human mind processes information (Jörges, 2013).

3.2.4 Different approaches

Several approaches to code generation are described in the literature. The practical implementation of an approach for an organization may come with unlimited choices. The main categories into which code generators can be divided are those who produce complete code and those who generate structures or skeletons that has to be finished by a developer (Jörges, 2013). Different tools for code generation and different approaches can result in different completeness of the code, also depending on the details of the model (Zhang and Patel, 2011). As an example, a given class diagram that expresses only the static information of the domain can be generated as the corresponding code skeleton of the class. A more complex generator and more information could generate most of the system behavior (Brambilla et al., 2017).

Round-trip engineering is a technique which aims at synchronizing model and code. This requires synchronization from both model to code and code to model, where changes in any of those levels are automatically propagated into the other level (Jörges, 2013). Round-trip engineering is a result of wanting to always keep models up to date with code, instead of abandoning models because of the resources it takes to keep them consistent (Selic, 2003). A challenge in all software development process is keeping artifacts such as documents, models and code up-to-date with each other. These tasks are often manual, time-consuming and error-prone, which round-trip engineering could be a solution to (Cicchetti et al., 2016).

Another code generation technique is partial generation. This means that code needs to be completed manually to receive a fully functional system. For software developers, this is more of a supportive code generation technique where you can receive skeletal code structures (Selic, 2003). This method means that there is no single source of information for a system and could lead to confusing inconsistency (Brambilla et al., 2017). The model contains some information about the system, and the code contains the same (that has been generated from the model) and even more added by the developer.

Round-trip engineering and partial generation can be combined. This means that changes in the parts that were generated are synchronized back into the model. Keeping model and code consistent means that code that has been generated from the model can’t be over-written or deleted. If code has to be added that could possibly have been modelled,

it should be modelled. It is forbidden to add code that can be described in the modelling language (Jörges, 2013). The solution means that the developers are only allowed to change the code for which they are absolutely responsible. Practically this could be solved by defining protected areas which can only be manually edited by a developer (Brambilla et al., 2017).

Full code generation is another code generation technique. It is an alternative to round-trip engineering and partial generation and means that any change in the system has to be made on model level. Modifications and hand-written code is forbidden and code generated from the models is fully functional without changes. Because the code is never modified manually, the code generator can over-write it carelessly and safely (Jörges, 2013).

Using the idea of full code generation, there is also a possibility to generate complete code for some parts of a system, like some subcomponents, while using other approaches for the rest of the system (Brambilla et al., 2017).

3.2.5 Testing

Software models can be used for early identification of errors by building test cases and running them against the model. When testing early in the life cycle, effort of rework things later in the process will be reduced. Glass (2003) says it is 100 times more expensive to fix errors in production than in the earliest development phases.

3.3 Tailoring a Development Method

Imagine the case where an organization with no modelling or code generation experience tries to adopt a full model-driven approach. The change will have a lot of direct and indirect effects in the organization, its processes and its teams and their products (Brambilla et al., 2017). It becomes critical for an organization to know where, how and when a model-based or model-driven concept should be introduced to make it a success. This requires an understanding of the structures, businesses, products and processes of an organization as well as understanding of the desirable method. The method must cover the socio-technical requirements of the organization (Whittle et al., 2014). Help from a modelling or code generation consultant or specialist in the adoption process could help in the change process. A decrease in both productivity and quality in the first project where a new approach is tried can be expected. This doesn’t mean the approach should be avoided. A successful adoption also requires a progressive and iterative approach combined with a motivated organization (Liebel et al., 2014). Benefits can be achieved first after the learning curve is overcome, which means that cost and time estimations for business-critical projects must be made with knowledge of these circumstances (Glass, 2003).

3.3.1 Process requirements

Franky et al. (2016) describes how to implement a model-driven environment in a software development organization that works with enterprise software. Some findings from this paper describing the requirements on the existing assets of the software organization may be applicable to the embedded software domain too. Implementing a new method like model-driven development requires according to Franky et al. (2016) a certain standardization and possible large-scale reuse of components. New method success is achieved by standardizing the architecture, enabling large-scale reuse of the future software models. Mattsson et al. (2012) thus state that preparing for the new modelling approach, by developing a reusable model-driven or model-based environment, requires a lot of effort and can decrease performance. In software development at an organizational level, similar software components tend to be frequently recurring in products. If each software product has a different architecture, it is hard to integrate existing components and reuse artifacts. There is thus a need to choose and establish a standardized architecture to enable the reuse and integration of already built components. This can be done by modularizing and multi-leveling the software into different architectural layers.

Franky et al. (2016) furthermore describes how existing, already built software components can be refactored and adapted based on the chosen architecture. Aligning the refactored components to address different architectural layers enables the reuse of them. Bassi et al. (2011) describes more specifically how models that were developed at a detailed level can be translated to another formalism and reduced in detail to make them reproducible.

Mastering the complexity of large and at the same time detailed models on different levels can be done with hierarchical models, which means supporting that models are embedded into each other. Hierarchy also enables showing only the parts of the models that are of current interest or relevant at a specific time or occasion (Jörges, 2013).

3.3.2 Competitiveness

In the field of continuously changing development standards, software development organizations have to utilize software development technologies as well as possible (Trendowicz, 2013). Cicchetti et al. (2016) has shown that there is a trend in increasing complexity combined with less time and budget for software projects. Facing this, organizations have to avoid all sources of inefficiency and mistakes. In fast changing markets, this means that there is a need for fast adaption and quick changes. Agile techniques have helped software development projects to adapt to these circumstances (Cicchetti et al., 2016).

As in all teamwork, software development teams are also challenged by cultural and communication gaps. Customers, developers and managers may have different mindsets and backgrounds which can lead to problems if not handled correctly (Jörges, 2013). For

a software corporation, organizational and human management aspects of software development can be critical for competitiveness. In addition, technical complexity has to be managed too. Individual techniques, methods, tools together with development processes, architectural issues, quality management and improvement and certification makes competitiveness an overall complex issue (Trendowicz, 2013).

All aspects of competitive software development have in common that they can be critical for the outcome of software projects. All possible risk factors have to be taken in consideration when working with evaluations and improvements of software development processes (Clarke and O'Connor, 2012).

3.3.3 Embedded systems development

An embedded system is described by White (2011) as a computerized system that is purpose-built for its application. They dominate the type of computer systems today and can range in size and complexity from small platforms to large distributed systems of myriads of interactive nodes. Embedded systems often have a tight integration between mechanics, electronics and information processing, where software implements the complex system functionalities (Crnkovic and Stafford, 2013).

Embedded systems software development can be difficult since it is limited by the hardware constraints but still has to fulfill the intended needs of the system (White, 2011). Embedded systems characteristics often involve timing and performance properties and a limited amount of for instance energy and memory resources. Systems also have to be reliable and safe (Crnkovic and Stafford, 2013). The management of increasing software complexity and advances in hardware is described by Crnkovic and Stafford (2013) as a challenge in the embedded systems domain.

3.3.4 Agile development

Agile project management has its origin from saving project money. Traditional project management approaches like waterfall methods often emphasize the importance of detailed planning. These standards have shown to be limiting for software development since the conditions and requirements often change quickly in real project settings (Cervone, 2011). Traditional software project methods provided very little or no flexibility at all, but the nature of software development involves requirements and specifications that can never be definite (Biju, 2008). The idea of agile software development is to answer to a need to develop software in an environment of requirements that are changing suddenly and still make the process quick (Greer and Hamon, 2011).

In agile methods, people are the key to project success (Biju, 2008). Agile teams are small and self-organizing, where knowledge is tacit instead of being based on extensive documentation. The iterative process is common to all agile methods, where continuous improvements and continuous integration is emphasized as well as frequent releases (Greer and Hamon, 2011). There are different approaches to agile methods, where some

basic concepts state the core principles. “The agile manifesto” state the core concepts of agile project management (Cervone, 2011):

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

These four principles are widely used and well established, but come in different modifications (Cervone, 2011). One type of agile project management is Scrum, which is well aligned with the values and principles of agile methods.

Both modelling methods and agile approaches have been widely adopted in different versions in industry for some time. Zhang and Patel (2011) describes how combining model-driven practices and agile methods has the potential of reducing software development cycle times and increasing productivity and quality. The key is in avoiding shortcomings of both methods as well as maximizing their benefits. Practically this means enabling mistake free development by code generation and at the same time achieving effective iterations from systems engineering to system testing. Franky et al. (2016) discusses that teamwork has to be supported by simple version control. Tools used for modelling must facilitate teamwork, so that dividing, sharing and incremental planning of work is not hindered by working at a model level.

Combining model-driven methods and agile methods means merging core concepts together. Zhang and Patel (2011) describes how this application of agile into modelling methods or modelling methods into agile is not likely to produce a short-term benefit, but after a while worth it for large projects with multiple releases. Development rates, improved product quality and shorter delivery cycle times are long-term benefits for the combinations of these practices.

3.4 Themes for Interviews

The themes found in the research overview have been grouped into general themes and sub-themes for an easier overview. The general themes and the number of sub-themes are found in Table 1. Themes will be used to make relevant interview questions to cover areas of concern.

Table 1. General themes.

General theme	Number of sub-themes
Code generation	7
People and team	4
Software challenges	7

The general theme of *Code generation* holds sub-themes closely related to the sphere of model-driven approaches with code generation. These are summarized by abstracting key findings most tightly related to code generation and modelling in the research overview, summarized in Table 2.

Table 2. Code generation sub-themes.

Sub-theme	Description	Source(s)
Design	Emphasize design decisions, making design easier	Whittle et al. (2014)
Testing	Model tests, generating test cases, easier testing	Labrosse (2007)
Modelling	Model creation effort	Chaudron et al. (2012)
Model combination	Combining high-level and low-level models, combining system and software models	Bassi et al. (2011)
Tool	Reliable tools for large-scale models and related tasks	Liebel et al. (2014), Jörges (2013)
Version control	Version control for models or code	Franky et al. (2016)
Training	Training time and cost, learning curve	Glass (2003), Whittle et al. (2014)

The general theme of *People and team* holds sub-themes closely related to people management, group work and how information is shared and communicated within a company or a software organization. These themes, presented in Table 3, are not specific to any kind of software development approach but can be discussed for all software development methods.

Table 3. People and team sub-themes.

Sub-theme	Description	Source(s)
Communication	Communication between people and departments	Jörges (2013)
Resistance	Will to change methods and processes	Whittle et al. (2014)
Handovers	Handovers of artifacts and information between groups	Glass (2003)
Agile	Agile methods	Zhang and Patel (2011), Cervone (2011)

The general theme of *Software challenges* holds sub-themes closely related to typical challenges that all or many software organizations deal with. These themes are not specific to any kind of software development approach but can be discussed for all approaches. The sub-themes are listed in Table 4 below.

Table 4. Software challenges sub-themes.

Sub-theme	Description	Source(s)
Quality	Number of bugs and errors of generated code, readability and understandability	Jörges (2013), Brambilla et al. (2017), Zhang and Patel (2011)
Reuse	Reuse of artifacts, architectural layers, refactoring	Franky et al. (2016), Bassi et al. (2011), Mattsson et al. (2012), Labrosse (2007), Marincic et al. (2013), Brambilla et al. (2017)
Documentation	Documentation efforts and needs	Brambilla et al. (2017), Cicchetti et al. (2016)
Synchronization	Synchronizing system design and implementation	Brambilla et al. (2017)
Release	Effects on cycle time and release time	Zhang and Patel (2011)
Maintenance	Maintaining software	Chaudron (2017)
Standards	Safety-critical software, domain standards	Jörges (2013)

Except from the above themes and sub-themes, there are some process choices that are discussed in the research overview. These concern possible approach or method choices and are presented in Table 5.

Table 5. Approach or method choices.

Method	Span	Source
Modelling paradigm	System/software	Bassi et al. (2011)
Approach	Round-trip/partial/full	Jörges (2013), Brambilla et al. (2017), Cicchetti et al. (2016), Selic (2003)
Abstraction level	High/low	Chaudron et al. (2012), Jörges (2013), Bassi et al. (2011), Glass (2003), Brambilla et al. (2017)
Extent	Part of system/Whole system	Brambilla et al. (2017), Selic (2003)

4. Case Study

4.1 Planning

There is a need for covering the whole systems engineering and software development process where model-driven methods and code generation is or could be relevant or touch upon. This means covering the process from incoming requirements to internal releases, which is the decided delimitation for this case study.

Understanding what areas are relevant to dig deeper into later, preparation meetings and workshops were held. Those 3-4 occasions of different themes served as getting to know the company and the department of interest on an overall level. There was also a chance to take notice of particular subjects of interest for the organizations such as problems, main focuses, bottlenecks and similar for the upcoming personal interviews.

Being able to completely map and analyze two different processes of systems engineering and software development in different parts of an organization, finding key persons to interview is a good start. Unstructured meetings or interviews were held in the organization to get support in finding key persons for upcoming interviews.

Later on, in the longer personal interviews, all interviewees had the possibility to propose other candidates that they thought could give more insight in the interview questions and the particular subjects of the interview. This could be proposed by the current interviewee at the end of the interview when they could pinpoint who had the greatest insight in the discussed matters.

In understanding how modelling and code generation are used within a large company, Heldal et al. (2016) states that there is a need for having several interviews to get a clear picture of methods, processes and techniques. The author has interviewed both systems engineers and software developers in both product areas. In the next section, motivations of each choice of interviewee is described.

4.2 Interviewees

The total number of interviewees at Saab are nine. Their belonging to product area A or B is presented below in Table 6. The idea is to have evenly distributed interviewees at each product area. The product areas have other names internally but will be referred to as product area A and product area B in this thesis.

Table 6. Total number of interviewees.

Product area	Interviewees
A	5
B	4
Total	9

Within product area A, there are five interviewees. With one of the interviewees, there was four unstructured meetings. This interviewee is named with a nick name in Table 7. With the four other interviewees in product area A, semi-structured personal interviews were held. These will have nicknames in the thesis according to Table 8. All nicknames are created with a formula of X-YYZ. X is replaced by A or B depending on product area A or B. YY is replaced by SE for system engineers and SW for software developers. Z is a numbering if there are more interviewees with the same role in the same product area.

Table 7. Unstructured meetings.

Nickname	Role
A-SE1	Systems Engineer

Table 8. Semi-structured personal interviews within product area A.

Nickname	Role
A-SE2	Systems Engineer
A-SW1	Software Developer
A-SW2	Software Developer
A-SW3	Software Developer

The interviewees in product area A have been chosen carefully to cover broad aspects of the development process and all internal work with software. Their roles are described in Table 9 together with a motivation for interviewing them.

Table 9. Interviewee roles A and motivations within product area.

Nickname	Motivation
A-SE1	A-SE1 is a systems engineer within product area A who has been responsible for introducing model-based systems engineering at Saab. The interviewee has good knowledge of the modelling field. The system modelling process has been discussed during 4 workshops or presentations with this systems engineer.
A-SE2	A-SE2 is working both as a systems engineer with modelling responsibilities and as a systems architect. A-SE2 has a good understanding of what can be improved in the software development at Saab as well as with the existing modelling approach within product area A.
A-SW1	A-SW1 is a software developer that is part of a Scrum team in product area A. A-SW1 is now moving towards a systems engineering role and has good insight in the communication, deliveries and information that is transferred between the systems engineering department and the software engineering department at Saab product area A

A-SW2	A-SW2 is a software developer and scrum master in a software development team within product area A. The interviewee can give good insights in the software development process, its strengths and weaknesses.
A-SW3	A-SW3 is a software developer and product champion. This interviewee plans and prioritizes work in projects concerning future software platforms and reusable components within product area A. The interviewee can talk about the current state of software development processes, what changes or investments in the internal software development processes are planned and why these are important.

Within product area B, four interviews were held. The interviewees will have nicknames in the thesis according to Table 10.

Table 10. Semi-structured personal interviews within product area B.

Nickname	Role
B-SE1	Systems Engineer
B-SE2	Systems Engineer
B-SW1	Software Developer
B-SW2	Software Architect

The interviewees in product area B have been chosen carefully to cover broad aspects of the development process and all internal work with software. Their roles are described in Table 11 together with a motivation for interviewing them.

Table 11. Interviewee roles and motivations within product area B.

Nickname	Motivation
B-SE1	B-SE1 is a systems engineer within product area B that, unlike all other system engineers in this part of the organization, has chosen to model the systems that he designs instead of using documents. The interviewee has a good insight in how systems modelling can be combined with code generation and why a modelling approach can be advantageous.
B-SE2	B-SE2 is a systems engineer within product area B which uses documents to do systems design, and is not concerned with modelling or models at all in this role. In a previous role, B-SE2 worked with model-based systems engineering in product area A and can therefore compare the approaches. The interviewee can give clues to why modelling approaches should or should not be limited to one part of the organization or to only one group.

B-SW1	B-SW1 is a software developer within product area B which works with code generation. The interviewee can give insights in this approach to software development as well as the process, learnings, advantages and disadvantages with code generation.
B-SW2	B-SW2 is a software architect and software developer within product area B concerned with more overarching software questions. This interviewee can give valuable insights in the chosen modelling and code generation approach.

4.3 Interviews

The reason for choosing interviews as primary method for gathering information is that the information required for the results are not available in documented form. The material for the thesis must be found from people that answer questions relevant for the results. Since questions related to this thesis are somewhat complicated and require supplementary questions and discussion, personal on-site interviews are strongly preferred (Eriksson and Wiedersheim-Paul, 2011). Disadvantages with personal interviews could be non-anonymity of the interviewee which limits the answers to what is comfortable talking about (Eriksson and Wiedersheim-Paul, 2011).

Interviews can be structured in different ways, and the most suitable approach for this case study is a semi-structured interview. The sub-themes from the research overview form main questions but supplementary questions are important to get most of the situation (Eriksson and Wiedersheim-Paul, 2011). Main questions are meant to start talking about a subject in a very open way, without asking detailed and focused questions. The purpose is for the interviewer not to lead the interviewee into conclusions but to see where the subject takes the interview.

One important aspect of interviews is how to handle the resulting material. Preferably, personal interviews are recorded to enable detailed walkthrough of the material. Otherwise, meticulous notes have to be taken, which can cause inattention of the interview (Ejvegård, 2009). All the interviews are recorded with sound only.

All interviews started with a presentation of the author, the aim of the project and how the material was supposed to be used (Ejvegård, 2009). All interviewees were informed that they will be anonymous as respondents. The interviewees were offered anonymity down to product area belonging and role description.

Process mapping focus areas

All interviews contain process mapping themes. The process mapping interview has a focus on identifying:

- Artifacts in processes
- Activities in the process

- Key persons, roles and their skills and competences
- Formal and informal communication
- Verbal or documented information

Each interviewee has the chance to give unique input in different stages of the development process based on their roles, tasks and experiences. The appendix contains interview questions for process mapping.

Code generation focus areas

From the research overview general themes and sub-themes that cover specific areas of concern are summarized and form the expansion to process mapping interviews. Themes cover aspects related to the possibilities with modelling and code generation, advantages and disadvantages with a socio-technical perspective as well as the possibilities of extending modelling approaches with a code generation approach. The appendix contains code generation interview questions.

4.4 Data Classification

Data collected from the interviews will be analyzed through classification (Ejvegård, 2009). In this case, classification will relate to general themes and sub-themes presented in Section 3.4. Transcripts from the interviews are read carefully. Answers and comments that match a sub-theme will be labeled and classified as belonging to this sub-theme, like in Table 12 below. When all data is divided into what sub-theme it belongs to, data from one sub-theme is collected across all material and summarized into the results.

Table 12. Data classification.

Transcribed material	Interpretation	Key findings	Sub-theme	General theme
”One of the advantages of working model-based is that you get a lot of pictures. We use them to show our customers and others how it works” (Interviewee X)	Models are good for visualizing system functionality both externally and internally	Visualize, show, display, understand	Communication	People and team

Classification is made with the intent to leave as little data unclassified as possible and to give findings from interviews coverage from literature about the same theme (Ejvegård, 2009). Regarding data that cannot be classified as belonging to one of the sub-themes, this data will be used as a description of the development process or be regarded as not relevant for the scope of the thesis.

5. Interview Findings

Sections 5.1 and 5.3 give answers to **RQ1**. This research question addresses the software development processes and how modelling and code generation is used within the different product areas A and B. Here, both product areas are discussed with the sub-themes of *software challenges* and *people and team*. Only product area B is discussed with the sub-themes of *code generation*, since it is only applicable to this area.

Sections 5.2 and 5.4 give answers to **RQ2**. It is meant to address challenges and advantages that the different product areas face.

The sections are product area specific, where 5.1 and 5.2 present results from product area A and 5.3 and 5.4 present results from product area B.

The projects within product area A are often a few to many years longer than projects within product area B. The technology is more advanced which requires more specialists and teams with different expertise. The products within product area A have a wider range of interfaces towards other systems, which makes the products and problems more complex. Product area B develops sub systems for other systems, which makes them less central and requires less interfaces.

5.1 Product Area A: Development Process

A new development project within product area A is initialized when customer specifications for a new project are delivered to the system engineers. Requirements and expectations from the customer as well as requirements and expectations internally are used to specify what functionality is needed for a system. The system engineering work is model-based and models are created with SysML in a modelling tool. System engineers at Saab have created the system modelling methodology on their own, with guides and rules in order to ensure consistence and coherence. At the stage of working with requirements and basic functionality, the system engineers make the first attempt to model the system. The system is described with anatomies (what the system should be able to do) built up by different system capabilities (what functionality is needed to do so). Which capabilities of a system to deliver at what time is specified in planning documentation that system engineers use to plan how the system model should be prioritized and incrementally developed and extended.

The system model is divided into units called function blocks. Function blocks consist of system components, smaller units than function blocks. System components must be deployed either on software or hardware. When a function block in the system model is going to be implemented, the model is used to generate documentation on the function blocks. Specifications of function blocks are delivered to the software developers in paper form, called function block design descriptions.

Software developers use the descriptions of function blocks to understand how the software is supposed to work and plan the agile development. If any mistakes on model level has been made, the system engineers update the system model. New function block design descriptions are generated and the software developers are notified about new documents to read. The overall development process is visualized in Figure 2.

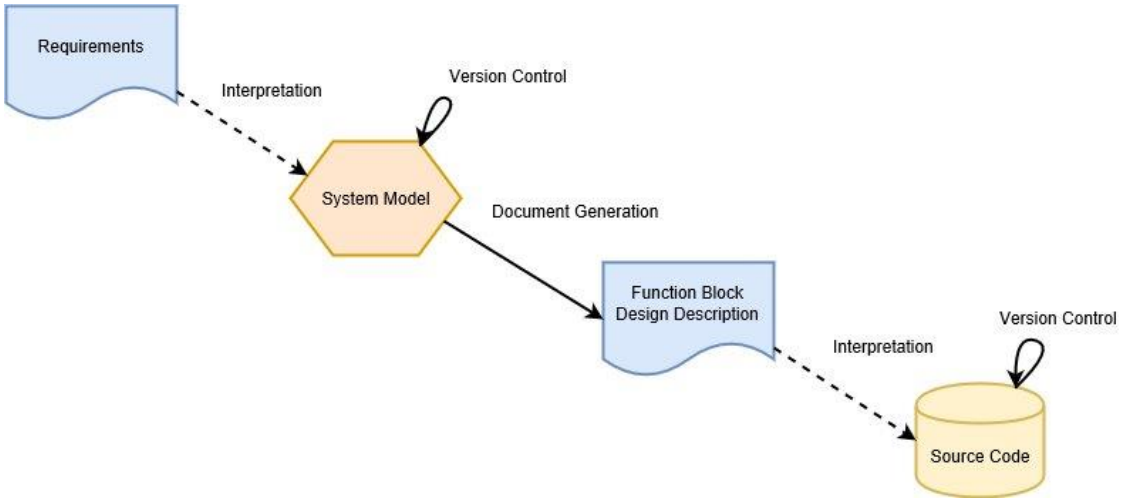

Figure 2. Development artifacts in product area A. Dashed line means that artifacts are related but not connected. Filled lines means that artifacts are both related and connected.

The software developers work agile with a Scrum methodology. They work in two or three-week sprints where each sprint is planned in the beginning and evaluated in the end. Releases are made with support from automation architects, who automates some of the release tasks. Sprints are parts of longer development periods called increments, which are meant to implement larger chunks of system capabilities.

In addition to teams working specifically towards products within area A, there are also teams with responsibility for developing and maintaining a software platform that is shared between several product areas. The goal is to enable reuse and maintenance more effectively by having one platform base serving all products.

5.1.1 People and team

Communication

The part of the value chain that involves an update of the function block design description is very critical, as every stakeholder needs to review the update. Interviewee A-SW3 describes how every team concerned from different disciplines, software developers as well as the system integration and verification teams have to read and interpret specifications.

The function block design description serves as communicational artifact between the system engineers and the software developers. This is supported by inviting system

engineers to increment or sprint planning meetings. The system engineer invited has a presentation on the function block relevant at the time, and can answer questions and introduce the software team to the task (A-SW2). A-SW1 highlights one of the reasons it is good to have a system engineer attending.

A-SW1: ”*We [software developers] only look in the generated documents, not in the system design [the model] itself. Sometimes the generated documents are not enough for us.*”

A-SW1 furthermore explains that the information provided for the software developers is not always clear or understandable. When comparing the interviews with A-SW1 and A-SE2 it shows that there is a grayscale in the exact handover point between the system engineers and the software developers. The artifact of the function block design description needs to be supplemented with supportive activities like meetings and presentations, which are exceptions of the formal handovers. As systems engineering is a very broad field of expertise there will be situations when both parties have to push the boundaries of their domain knowledge.

When developers start implementing the software, it happens that they realize that the solution needs to be redesigned. Software developers reach out to system engineers by e-mail or in person, and a discussion about the solution starts. According to A-SW1, the discussion can lead to both better understanding of the model for the software developers, or a need for system engineers to illustrate the functionality another way. It happens that the problem needs to be solved with another design.

A-SW2 mentions that communication about how things work is very common in the software team. Primarily the software developers seek each other’s help and only occasionally try to approach someone from system engineering. The solution to a problem can often be figured out by the development team internally with some collaboration.

Using a system model to create function block design description helps every team to get an overview of the system and how things work, which is mentioned as the greatest advantage related to systems modelling and communication (A-SE2).

Resistance

The resistance to change is hard to evaluate and measure. During interviews and small talk at the company, the opinions differ. The most recurring comments have not to do with modelling or code generation itself, but the amount of time and effort needed related to changing work behavior, processes and having to learn something new. One interviewee mentions a thought of software modelling as messy and that if there is time pressure people will not follow the guidelines (A-SW3). A-SW1 thinks of modelling tools as more complicated than usual IDEs for software development.

A-SW1: *“It feels a little bit boring.”, “I feel like it’s more complicated to work in a modelling tool and do programming instead of in a usual IDE for the specific language.”*

Handovers

The handovers from systems engineering to software development has been discussed to some extent under the sub-theme of communication. The handovers are made by delivering generated function block design descriptions to the software teams. The delivered artifact is a document with its source in the system model. If the system design is updated, there is another handover after documentation has been re-generated. The software team is notified by email about a new version of the function block design description.

Agile

The system engineers have split the system and also the individual work into function blocks. The person responsible for each function block works with this part of the system, plans and delivers function block design descriptions to the software development team using input from stakeholders.

The software development team works with Scrum. The product champion has an overview on upcoming milestones, when different functionality should be released and how to prioritize work. The product champion specifies large chunks of work, called Epics, for each upcoming increment. An Epic can be a large function or something that spans over many software modules. The development team has a planning period when Epics are decomposed into Stories. Stories are smaller and more distinct parts that can be planned and time estimated. When decomposing work into Stories and getting an overview of what to do, there can be activities needed to support planning. These can be rig tests, reading documentation, integration and verification results and talking to the designer. A lot of work is also made to investigate what's already implemented and how to proceed with that.

The development is incremental and done in sprints. Each sprint starts with sprint planning and ends with a retrospective. During sprint planning, the team tries to estimate how much work they can do. In the end of a sprint there is a retrospective where the team evaluate how they have worked in the sprint. Approximately once per month there is a demo in the end of the sprint, when the team thinks that there is something to demonstrate. Every second sprint there is a risk meeting where the team summarizes possible risks related to the software that the team are responsible for.

5.1.2 Software challenges

Quality

There is not enough information from the personal interviews to measure or discuss human errors and bugs in the development processes. It is hard to measure the data in terms of code quality in product area A compared to product area B, since the

development teams measure different things and present different data. The products are also developed with unlike conditions like different people, standards and functionality.

In the research overview it is mentioned that hand coding can have more bugs and be error prone due to human factors and variations among developers.

Reuse

Within product area A projects there are two entities of common and generic software that developers can use. Common software has diverged in different versions for different projects. This means that what is supposed to be common is not common because it exists in variants for different products. A-SW3 is the product owner for a project that is re-inventing reusability within product area A.

A-SW3: *"We want to have a bottom layer, like a groundwork that is common to every project and doesn't diverge. Then we want to have common system components."*

A-SE2: *"What we are doing now, both with the architecture and with the software, we imagine will be used in many future projects. So, we are putting a lot of effort into thinking about how to reuse both models, specifications and code."*

A-SE2 mentions the idea of having a reference model, where a new system can be created by picking and choosing from the reference model. The reference model is not related to a specific product but can be composed to new models for specific products. The composed models can be completed with product specific parts.

A-SE2: *"It would be really great to have one of those."*

The idea is creating system components that are reusable, because system components are frequently recurring in products. The interviewee describes the link between the reference model and the reusable software plans.

A-SE2: *"If I choose to reuse a system component from the reference model, the code should already exist."*

Documentation

A normal sized function block in the system model within product area A results in around 200 pages of documentation when generated as papers to be delivered to the teams. The function block design descriptions are detailed and describe the software to a large extent. The documentation consists of both text and pictures, where pictures are sequence diagrams, block diagrams or state diagrams. However, the resulting source code has no specific documentation describing the implementation other than the code itself and comments in the files.

A-SW2: *"The code is the documentation."*

There are no external or internal requirements within product area A to deliver implementation documentation. The comments in the code can be used to understand it, where there are internal rules on what should be included in file headings to make it understandable (A-SW2). This include descriptions of what methods do, what they return and descriptions of parameters.

Release

Releases are supported by automation architects and release managers that work on automating and optimizing release tasks. Within product area A, special teams have the recent years started working with Continuous Delivery and DevOps to speed up, simplify and increase the quality of work. Tasks related to writing release summaries and release notes are automated, which is fantastic according to A-SW1. Several development and release activities in the development pipeline are being reinvented. Integration and verification teams receive the releases together with information about the release and test the new software. The software developers get feedback on what's working well and what's not (A-SW2).

At the end of an increment the software teams write formal documents to accompany the software release. One is a version description that describes the content of the release in terms of new features, known errors and fixed errors. The other is a product specification which contains information on how to build an exact copy of the software that has been released.

Maintenance

The teams are trying to make every developer follow some basic design rules and setting common software architectures. By making components look similar, the idea is that general components will make maintenance better by enabling easier changes. Teams in product area A try to enable easy maintenance by stressing the need to comment the source code.

A-SW2: *"We try to not make spaghetti code [unstructured code] that is hard to understand. There are some things that you have learnt gradually, like not making too large functions, and split things to smaller classes."*

A-SW1: *"Comments in the code."*

Comments in the code makes it easy for a person that hasn't created the software to understand it and make changes (A-SW2). The team also tries to have good function names and not introducing a lot of dependencies that are unnecessary (A-SW2). To support bug fixes there are log prints that explain errors and where fixes are needed (A-SW1).

A-SW1: *"We also do "*how-to's" *for ourselves, where we describe difficult functions, paint diagrams and describe flows."*

Synchronization

In an incremental approach there is a challenge in identifying what parts of the system model should be implemented in the current increment. The definition of the system design takes entire uses cases or requirements into account whereas each increment gradually delivers parts of the entire requirement. This means that for every update of the system design, a mapping must be performed into what should be delivered in every increment.

Interviewee A-SW1 explains that after the team receives a new version of the system design, the team has to overview what they have already implemented. The new version can both consist of new detailed design of new incremental work, but also changes to parts that are already released.

A-SW1: *"The system engineers can update it whenever. They just decide "*let's rename this part and move it over there!*" and we still have the old design implemented in our code."*

The interviewee describes that their work is a lot of refactoring. The other way around, it also happens that the system designers have to re-think the design. What they plan isn't possible with what's already implemented (A-SW1).

The idea is that the system design should be ahead of the implementation (A-SE2), but it happens that the developers are ahead of system engineers or implement parts they need without having it planned. At this case, the system engineers realize in the model what the developers explain that they have implemented.

Standard

Projects in product area A have no external safety-critical software standards to follow. Still, some standards are looked at to ensure good software standard, without external pressure.

5.2 Product Area A: Challenges and Advantages

System engineers use models to plan and design the system. The software developers receive documentation generated from a system model designed by system engineers. Software developers within product area A do not use code generation or model-driven approaches in their work.

The interviews showed that the idea of model-based systems engineering in product area A is to gather information, plan and design the system. The intention is not generating documentation for software developers. Presenting the relevant information in a good way for software developers to understand and work with seems to be something that is hard.

Synchronizing the design of a system and the implementation is a lot about mapping and investigating. The software developers can be surprised with changes that affect the already implemented software, because of the challenges to do two incremental processes in parallel. They describe their work as a lot of refactoring and looking through the implementation and adapting it to new designs. System engineers are not always familiar with what is already implemented. Both system design and software development work is done incrementally in some way and both divisions can run ahead of each other and affect the work of the other.

There are teams focusing on improving the software development methodologies within product area A. The DevEnv team introduces Continuous Delivery and DevOps by optimizing and automating release and development tasks. This is focused on smoother handovers, avoiding time consuming manual tasks and overviewing the whole deployment pipeline. A platform team works with effective reuse and maintenance of software. They are trying to make a successful reuse plan applicable to both models, specifications and code. The idea is to have some kind of reference model where system components can be found and fetched. If a reusable module is needed, related code should already exist and be ready to use. The reuse requires a lot of effort according to involved persons, but if successful it is applicable to many future projects. Reuse is not only necessary to make each new project faster, cheaper and easier but to enable future maintenance. Today the teams enable maintenance by writing structured code, commenting the code and writing guides for themselves with descriptions and diagrams.

The above discussed challenges and advantages are summarized in Table 13, where a green plus marks enabling or successful factors within product area A. A red minus marks weakening or challenging factors within product area A. All factors are related to model-driven approaches or methods.

Table 13. A summary of challenges and advantages within product area A.

Factor	Description
+	Successful approach in model-based systems engineering
+	No external requirements on documentation
+	Continuous Delivery and DevOps investments
+	Verbal communication and involvement
+	Scrum methodology
+	Reusability investments, platform team
+	Will and ideas on how to improve development
+	No RTCA levels or standards
-	Documentation can be hard to interpret
-	Refactoring and changing implementation
-	Challenges in synchronization of plans, design and implementation
-	Signs of resistance of large change processes
-	Maintenance challenges
-	Existing software platform

5.3 Product Area B: Development Process

In product area B, system engineers plan the functionality to be solved by software and hardware, which can be for example mechanics or electronics. The system design describes what a system should be capable of doing, but not specifying if this is supposed to be handled by software or hardware. The products in product area B are planned on a system level by system engineers working document-based, which means not with modelling system functionality.

The basis of system design and architecture are system requirements obtained from higher level, like an external order for a new system or an update of an old system. Products within product area B are to some extent based on similar concepts and are alike in terms of system functionality and design. When requirements for a new system are received, they can be matched with existing system specifications and the specification can be modified to cover all requirements. The system specification is then decomposed into subsystem design specifications, on a level of smaller units or components.

Software developers are handed documents describing preferred functionality of the system. The specifications are made with smaller parts (subsystems) in mind, describing the design of these smaller parts that make up the whole system. Based on this, the software team makes sure that the software fulfills the higher-level requirements by creating software requirements specifications. After reviews and discussions on a system level, the concept phase is over and the implementation phase can start. For software developers, this means creating a solution, designing and implementing the software. Software developers create software models described in the software modelling language UML. They use a modelling tool which enables modelling software and generating code. The tool has a built-in code generator, but developers in product area B use a code generator created by another Saab site and imported to the site in Järfälla. They use a full code generation approach where the models are the only source where software can be created or updated. The generated code is not altered, changes have to be made on model level. Models are version controlled, where each software developer has to ensure manually that no one is working in parallel on the same part of the model. The overall development process is visualized in Figure 3.

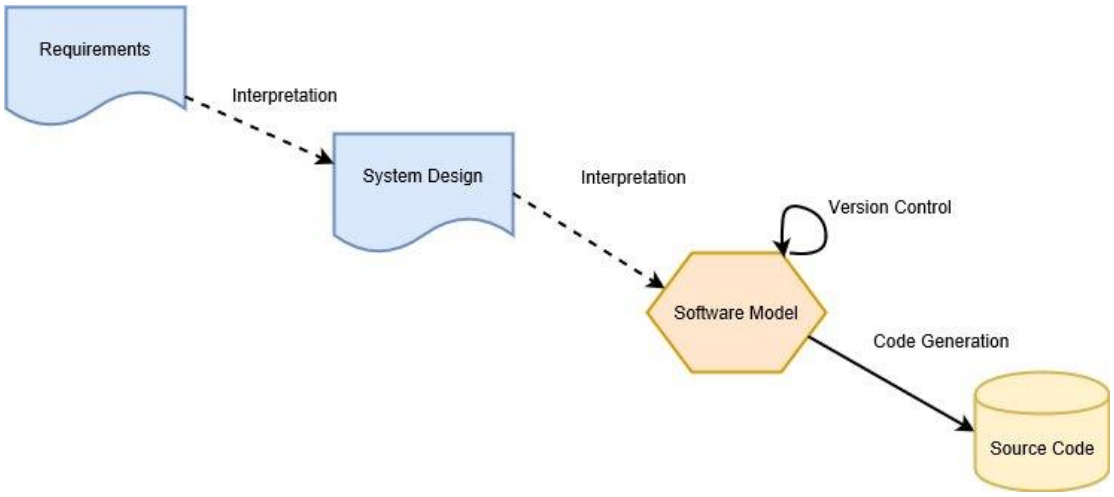

Figure 3. Product area B development artifacts. Dashed line means that artifacts are related but not connected. Filled lines means that artifacts are both related and connected.

Software models within product area B are structured in layers. The lowest level layer contains the most general models, general software components. The highest-level layer contains product unique models, very specific software components. There are five layers in total, where function or method calls can only be called from less general layers to more general layers, or inside layers. This enables reuse of components at lower levels, since the functionality of those are recurring in other projects. Building new software within product area B often contains of "picking and choosing" of preferred blocks on different levels and building the more specific, higher layer components, by making new models. When the overall model of the system in a specific project is done, it is mainly used to generate executable code and related documentation. Software models can also be created only for explanations, visualizations and documentation, not being used for code generation.

5.3.1 Code generation

Design

With code generation comes designing and building the system and/or software with models from which code can be generated. This has been discussed in the literature study as being a possibility for emphasizing design decisions. The interpretation of this is that design decisions made in the development process can be fulfilled more easily by modelling and code generation compared to hand-coding and non-model centric methods.

Product area B consist of real-time systems, which means that systems are acting under time-critical circumstances. System engineers and software engineers have described how systems that are developed need somehow special and extraordinary careful design to succeed (B-SE1, B-SW2). The act of modelling has been a great help for one system engineer in solving real-time aspects of the software and hardware (B-SE1). B-SE1 is the

only system engineer working with models and the interviewee chose to model the system design for his products, not sharing the approach with the rest of the system engineers within product area B.

System design and the software design are two separate and isolated processes within product area B. All documents are independent of other documents, where hardware has one design document, software has one design document and the system design tries to somehow bind it all together.

B-SE2: *"The problem with this is how tricky it is to see where functions are allocated … it is a detectives' work to find out if a capability of the system is solved by hardware or software"*

B-SE2 describes how the artifacts created in different parts of the development process are loosely connected to each other. The system design is not linked or connected directly to the software design.

Testing

Software models are not tested within product area B, it is the generated code that is tested. A testing tool is used to generate some parts of the tests but manual testing code has to be written too. Some of the tests that have to be run through and passed are static code analysis tests, code coverage tests, safety standard tests and unit tests. Unit tests are run for the whole repository, which means all blocks (components) on all layers.

When software modelling and code generation was introduced, it was discussed within product area B whether to test the model or the code. Testing the models could mean adding a separate test package in the models and testing directly in the models.

B-SW1: *"… if you are using a tool to automate tests, and don't want to double check the result, the tool must be internally qualified to be used … "*

To be able to test the models, the tool had to be internally qualified to prove that testing is done correctly. To avoid the rough work of qualifying tools (B-SW1), the developers within product area B chose to test the generated code instead of testing the model.

B-SW1: "*What we did was ignoring the model and instead examining and testing the generated code, otherwise it would have been very complicated*."

Modelling

Modelling in this context means the effort that is needed to create models. According to interviewee B-SW2, many people unfamiliar with software modelling seems to imagine that the work only includes choosing modules, shaping, dragging, dropping, and connecting them to each other. But modelling is not only "painting" according to the interviewee.

B-SW2: "There is no magic button to press and code is generated, we have to write a lot of code in the models."

Developers within product area B disagree with the common idea of that modelling is not coding. B-SW2 clarify that they get help with creating classes and similar, with support from the modelling languages and its visual elements. But hand-written code has to be added to the model, into the elements. The general thought of being given support from the modelling tool with creating elements and "filling them" with code seems to be that it is a great help and an easy way to work with software development.

When modelling is new in the software organization, the modelling efforts can be greater than normal coding because of training needs. Training time will be discussed later on, but a software developer describes how modelling efforts can also be affected by the need to setup modelling standards and develop processes for modelling.

B-SW1: *"In [project X] we had to do a lot from scratch … we had to develop new processes so the project took a longer time than usual"*

Model combination

This sub-theme is meant to cover the combination of models on different abstraction levels. Within product area B, models are used for software and not for systems. Theoretically a combination of models on system and software levels seem to be possible and preferable according to both system engineers and software developers within product area B.

B-SW2: *"I think that the system engineers should work with models. We could take over and work with that down to software level and generate our code."*

B-SE1: *"If you have modelled something, I can connect to you… Eventually I will end up in looking straight into your software."*

SysML as modelling language has its basis in creating units. A unit can refer to other units, where a unit also can be a UML unit. This means that browsing a high-level model, a system model and its units, you can end up in a looking at a software model unit (B-SE1). Units from another model imported or connected will be read-only. This helps the developer to know what parts that are allowed to do changes in.

B-SE1: *"You should be aware of, I think it confuses people, that you can connect a UML [software] model to a SysML [system] model if they are both made in [tool X]."*

B-SE2: *"You could fill it [the model] in piece by piece. I could deliver the model to software developers, and they will fill the details in."*

Connecting and combining models on different abstraction levels is something that is not done yet within product area B. The ideas seem to be many among the interviewees on

how to do it and the importance of doing it. According to B-SE1 there need to be clear rules on how to combine system models and software models. The inclusion of UML models in SysML models need rules on where software can be connected and how the connection should work.

B-SE1: "*If system engineers do one model and software developers do another model, they won't be the same. Communication is hard for them all, everyone is busy with their work, in the end of the day the models will not have anything to do with each other or there will be great differences. Therefore, I think it should be one model only.*"

B-SE1: *"You should not build separate worlds. Link the worlds [models] together."*

Tool

Tools central for software modelling and code generation within product area B are a modelling environment and a code generator. The modelling environment has a built-in code generator which is not used. Instead, a code generator developed at Saab is used.

An important factor when choosing a modelling tool was to give the software developer a good visual overview of the software (B-SW1). In classic development the overview is normally given by some kind of directory structure and the code itself. The possibilities with a good tool was getting the software class you are working on centralized with related interfaces surrounding, according to B-SW1. With the chosen tool, the developer can be given different kinds of overviews that are relevant depending on what kind of work you are doing. Another important factor according to both software developers for choosing a tool was the possibility to use small models to build a large model. Small models can be developed as independent modules and picked-and-choosed to create a large project (B-SW2). The tool is described as the one that fulfilled the criteria best.

The code generator used is developed at another Saab site and imported to product area B in Järfälla. The built-in code generator in the modelling environment was considered to be too expensive, why another alternative was searched for.

B-SW2: *"... if we use [tool X] without a code generator and apply our own code generator, it will be much cheaper."*

Not only was the price a factor for choosing the code generator developed at Saab. The built-in generator was not good enough for the purposes of product area B (B-SW1, B-SW2). The code that was generated looked strange, it consisted of a lot of information that was not relevant. Some of the things that are frequently developed in this domain, like state machines, were not readable.

There is not a lot of maintenance on the code generator today (B-SW2). What has to be done is updating and re building the generator when connected tools change version. There are some bugs in the code generator that developers are aware of, but these can only be fixed by the Saab site that developed the tool.

B-SW2: *"There are some bugs in it [the code generator] that you have to be aware of, but in my opinion, that's the case with all compilers."*

Some unexpected benefits have occurred with the tool choice. Developers within product area B can model functions that are not part of a class, which is a great bonus (B-SW1). It can be used to generate interrupt vectors and generate Main, which is not necessary when only creating UML models but when creating executable code.

Version control

Within product area B, models are checked into a version control tool. The tool is relying on artifacts that can be easily and correctly merged, like source code. Since the models are checked in and not the code, some problems have occurred. A model can't be worked on in parallel by two or more developers, which is described as problematic (B-SW1, B-SW2). If the original model has been altered differently into two versions, the merge of these versions is not possible.

The solution would, according to the developers, be splitting a project into small parts that can be handled and updated separately. What is required then, is extensive planning enabling this solution. Even if the model is divided into smaller parts, a successful solution would require locking the part of the model that the developer want to work on (B-SW2). Locking a model part would make it impossible for changes to diverge. It would not allow other developers to start altering the piece another developer is working on.

B-SW2: *"… And we can't lock files. We have to manually announce what parts we are working on, and no one can work on that part simultaneously."*

Manually announcing what part a developer is working on has caused troubles, or "conflicts" as in a version control language (B-SW2). The approach of manually announcing what part you are working on has caused trouble especially in intense periods, when a lot of changes are needed. These are caused by human factors according to B-SW2, when developers within product area B forget to mention what parts they are altering.

Training

It is clear within product area B that there is not much need for training in code generation, but in modelling. Code generation is described more like entering a menu and choosing to do code generation. What requires training is modelling software. For developers used to work object oriented, the understanding of models has not been a problem (B-SW2). It is the modelling tool and the internal modelling processes that a developer needs to learn to get started with code generation.

B-SE1: *"Allocate resources for training and education! The system engineers here [product area B] open [the tool] and they have no idea on where to click or how to do a certain task."*

B-SE2: *"I think that software people in general learn faster, a lot of people are used to UML and model-driven development isn't very different."*

B-SW2: *"Yes, I had to learn [the tool]. It's quite a challenge. I just did it, I thought it was very important to learn it."*

What is said about starting to learn modelling is that there is a need to do it in a real project, working on a real product. Playing around with the tools without a goal will not make learning faster (B-SE2). Having an experienced person around seems to be an important matter too. The developers within product area B have described that when learning the basics of modelling, it's more about understanding the syntax and understanding how to do it with the tool. For all learnings not to diverge, they highlight that there is a need for a common language, syntax and method that is established from the beginning.

5.3.2 People and team

Communication

Models are described to be a great help within product area B to visualize how software works, and its visual elements makes code easier to understand. One of the advantages described are the "pictures" or visualizations of software that modelling carries with it.

B-SW1: *"One of the advantages of working model-based is that you get a lot of pictures. We use them to show our customers and others how it works."*

The skill of modelling and visualizing your work has proven to be useful for other things than developing and communicating the code that is going to be produced. Having learnt the modelling language and making different kinds of diagrams, schemes and models for software has resulted in that skills in modelling are used for other purposes too.

B-SW2: *"If I'm going to visualize something for the documentation, I would paint it in [the modelling tool], not a painting tool. I've used the tools to create diagrams that are related to requirements too."*

When the teams have the same view and idea of the solution, it will facilitate that the results cohere (B-SE1). The model is explained to be a forum of collaboration within product area B, which simplifies work. Visualizing and modelling all types of work has furthermore meant finding possible failures before they are implemented.

B-SW1: *"... I visualized it [a state machine] and handed it over to the electricians. They looked at it and said that it was wrong and it wasn't going to work. I got that information*

very early in the project. Sometimes it's valuable to have the ability to explain what you are trying to do."

The software developers within product area B describes that they are involved in some system questions. They are normally involved in system questions concerning the interface between software and hardware, but are otherwise working on their own.

B-SW2: *"It's because system engineers are not working with models. If they would work with models to an extent that we do, it would be different."*

Systems engineers within product area B are not often in contact with the models created by the software developers.

B-SE2: *"Well, they have never told me that "*you can see that in the model*", they use them for their own purposes."*

Instead of looking in the models, systems engineers are looking at the generated code. This could be happening when there are problem reports that the system engineers receive from production or integration units, and have to start investigating the source of a problem.

The system engineer that has tried modelling the system has realized the great benefits for communication. Using SysML to model the system with graphical elements gives the team good material to work with.

B-SE1: *"You can throw documentation on people, but that doesn't work very well. Now, people really fast can get a sense of what they are going to do."*

The visualization is a key for the system engineer to explain and communicate. The commitment from the team is also something that seems to have improved. The interviewee describes that when a plan is obvious and clear, it is easier to give feedback on it. It has happened that software developers have had input in how to do things, and system plans can be changed in an early stage of the development process.

B-SE1: *"My biggest interest is the dialogue. I want to get his competence [software developer] and mine combined."*

Resistance

It has not been described as a controversial change to start generating code. Software developers within product area B describe how it takes some time to get used to, but the general thought of modelling and code generation seem to be that it's good and helpful.

Modelling and code generation, tools and approaches, was chosen and introduced by software developers. Software developers within product area B are very strict in mentioning that they are actually still coding "like you usually do", but in the models.

They want to look at modelling and code generation as an approach to help the developer with some tasks.

Handovers

What is handed over from systems engineers to software developers within product area B are artifacts in the form of documents. These are used by software developers to plan, design and create the software models. It has been mentioned that the gap between system plan and implementation could be smaller if these worlds could be connected (B-SE1, B-SW1). There is no obvious connection between the documents describing the planned system and the models of the implementation, as there are no system models.

Agile

Software developers within product area B use a Scrum methodology. The development team has chosen to have one Story to be one software component. Stories can be planned and time estimated, where each software component is planned and developed to fit the internal software layers. The software phases are short and agile, but in a perspective of longer phases, the projects follow more of a classic waterfall method. Different tasks like design, coding and integration are isolated to a clear step-by-step process, because of customer demands on performing certain actions in order.

System engineers work document-based at product area B. Individually or in groups they plan specifications for the system that covers the requirements for the whole system. These are delivered to software developers. It has been described by the system engineer that has tried modelling, that being able to view the system with different perspectives helps planning agile or incremental work (B-SE1). A scenario that can occur in system or software development is that developers create or implement something that is not usable or can't be run.

B-SE1: *"It's like building a house without doing any groundwork. Or building a window before you have a wall."*

The modelling can help sequencing the work and make incremental work less risky. The system engineer means that planning well can reduce the amount of re-work that has to be made in a later development phase.

5.3.3 Software challenges

Quality

The code generator produces code that is easy to read. It doesn't have traces of who the developer is but still is readable like a developer could have implemented it (B-SW2). The comments are nicely inwrought and there are good file headings. This is described by the software developers as related to choosing to import a generator developed by Saab. It is customized to Saab's use and has code quality and looks according to this.

B-SW2: *"You can't even notice the code's generated from the model, because the code looks so good."*

The generator is described to be almost bug free, but with some known bugs.

Reuse

When software modelling and code generation was introduced within product area B, reuse of software was planned from the beginning. A lot of time and effort was put into creating a methodology of reuse, which would enable that modules in projects could be reused. It has been described that a lot of though was put into how to make reuse successful (B-SW1).

Software modules that are frequently recurring in products were divided into layers that reminded of the product structure. The highest layer are project specific modules while the lowest are the most general modules, visualized in Figure 4 below.

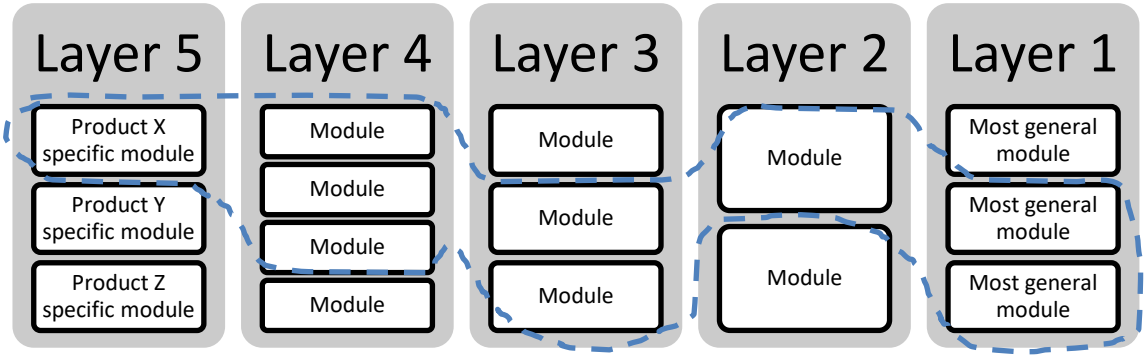

Figure 4. Software layer structure within product area B.

The product specific layer can use and be dependent on what's in lower levels, but lower levels can't do the same with modules that are more specific. This enables reuse of the lower level modules. It is described that the most general modules on the lowest levels are used in almost all products since the layer structure modules were released.

B-SW1: *"It's inspired by building LEGO."*

This kind of layer structure is not exclusive to modelling, it is a general method that software developers can use to enable reuse of software. Splitting the products and models within product area B into smaller parts was important for both reuse and to create a sustainable way of working with modelling and code generation. Reuse could be planned from the beginning when implementing software modelling and code generation. Planning required not only choosing a layer structure and the number of layers, but investigating the possibility of reuse of every single module created. The 5-layer structure was chosen because it was suitable for the products.

B-SW2: *"It's like a library of reusable components."*

Enabling reuse with a layer structure practically meant making separate units of the modules. A general component can be one unit. When creating a product model, units are connected. They are linked creating a large model for a certain product. But the unit can be used at the same time in other product models consisting of other units and other combinations. With a long time perspective, this means that the same software will reoccur in different products. Something that can save a lot of time and effort when changes or updates are needed according to a system engineer.

B-SE1: *"I can see every day how small variations in software leads to having one version of the software in [country X] and another in [country Y]. If something is wrong, you have to fix it at both places, in different versions."*

Documentation

Documentation is created from the models, which are the central development artifact in product area B. Documentation will always be up-to-date with the actual software (as long as new documentation is generated). Some of the diagrams, like sequence diagrams, doesn't have to be generating code. They are not for the implementation, only for documentation.

What has to be done to ensure the documentation looks good is inserting a lot of information into the models, to explain what objects are used for. Every object in the model has a description-field, and what is inserted there ends up as comments in the generated code. Almost all detailed design information has its source in information inserted in the models. If this is done carelessly, a lot of work has to be made with re-generating documentation and filling gaps. The tool can remind you of where information is missing or there is simply an ERROR-text in the generated documentation where information is missing.

B-SW1: *"Keeping the design description updated with the code is a serious problem. Documentation is always falling behind the code, but it isn't here. It's hard to not make it updated since the source is the same."*

Some documentation effort has to be made except from generating documents, where manual efforts are needed. Some of the information that has to be in the design description can't be generated. This can be an introduction or textual descriptions of software architecture and layer structure. This other documentation is put in a text document is merged into the design description together with the generated documentation. This is done with a template (B-SW2).

Release

Documentation efforts have been mentioned as something that can make the release smoother and also quicker, because no extra documentation work is needed when the software is created (B-SW2). What's also mentioned is how communication and

visualization advantages probably speed up the time to release, since bad plans can be corrected early (B-SE1).

Maintenance

Small variations in software lead to maintenance difficulties. Having introduced reusable components and a layer structure when introducing models has helped with maintenance. Corrections need to be done in one software and is applicable to all software, because of the layer structure and modularization.

Synchronization

Synchronizing documentation, planning, design, requirements and implementation is discussed in literature to be a challenge in software development. This is a problem that occurs when the development process consists of many separate artifacts. According to software developers within product area B, this is not a problem, because of code generation. They have one single artifact in software as source to documentation, design and code: their software model.

B-SW2: *"The advantage is that code and documentation never diverge."*

Everything has the model as its source: the software design, implementation and documentation. These will always be synchronized. What isn't synchronized is the system plan and the software implementation (B-SE2).

B-SE2: *"It's a great risk for us to have one specification and design that tells us how software should work, but never being sure we're correct."*

Standards

Products within product area B are developed with aviation standards like RTCA in mind. The levels explain how to develop safety-critical software. RTCA standards can be in different levels internally in product area B depending on product. The RTCA level limit the reusability to modules with the preferred standards (or better), since a module is not allowed to be reused in products with other RTCA levels. A module that is developed for a specific standard and fulfills these criteria can't be used in a higher-level project. How to handle reusability of modules developed with different standards is something that has to be investigated and worked with, according to developers within product area B (B-SW1, B-SW2).

The code generator that is built at Saab has been created with adaption to some level of standards. RTCA levels is not a software modelling or code generation specific problem but has to be solved regardless of approach choice.

5.4 Product Area B: Challenges and Advantages

Product area B uses a full code generation approach where all development tasks are made in the software model. The model is not connected to any system engineering artifact, it is constructed with documents describing preferred functionality. There is a joint view within product area B that combining models on different levels are both preferable and important. The advantages of using models and their gains in feedback, communication and visualization is today isolated to the software development process. Using one single model and connecting a system modelling sphere with a software modelling sphere is a wish for their own future process. They have stated that there need to be clear rules upon how to combine models and how to connect worlds together.

The communicational advantages of modelling are obvious for those who see, use or create software models. Models can be shown to customers and visualized to the team, which has improved commitment to the task and feedback between people. Models are a forum for great collaboration, which makes problems and solutions appear in time.

The approach of code generation and software modelling was introduced by software engineers within product area B when a new project was launched a few years ago. Tools were chosen to enable visualization of software to the developers and a code generator built at another Saab unit was chosen to make sure the generated code was readable, had good comments and fulfilled criteria on how software files should look.

Learning to use a modelling tool has proven to be the hard part of learning software modelling. It requires help from experienced persons and a clear guide with common languages, syntax and rules to help a team learning. Somewhat related to tool choice is also version control. The choice of working with a version control tool that is not good for merging models is an obvious problem. The manual locking method has proven to not be safe enough for many developers working in parallel.

The reuse plan with software modules in 5 layers was a clearly enabling factor for modelling and code generation to be successful. It required a lot of effort to plan reuse and create reusable models, but it has been used in almost all products since. The only limit for total reusability within product area B is that safety standards can be different for different products.

Since all reusable components are also in model form, they can apply visual and communicational advantages to all products using these. It also means that the same software modules from lower layers are in all products delivered with this software basis, an enabling factor for easy maintenance.

Documentation and implementation are always up to date with each other within product area A development projects. The software model is the implementation and the implementation is the documentation. Keeping artifacts synchronized and relevant is a

challenge in software development that product area B has solved with their approach to modelling and code generation.

All the above discussed challenges and advantages are summarized in Table 14, where a green plus marks enabling or successful factors within product area B. A red minus marks weakening or challenging factors within product area B. All factors are related to model-driven approaches or methods.

Table 14. A summary of advantages and disadvantages within product area B.

Factor	Description
+	Documentation efforts, automatic generation
+	Synchronization between software design and implementation
+	Visualization and pictures
+	Modelling expertise can be used for other tasks
+	Communication, a forum for collaboration
+	Quick feedback
+	Commitment from team
+	Tool support for various modelling tasks
+	Cheap code generator
-	Learning time
-	Tool qualification
-	Version control
-	Code generator dependency on another Saab site
-	Planning efforts
-	RTCA level makes reuse harder
-	Lack of a common modelling approach among divisions
-	Hard to trace functionality for system engineers

6. Discussion

*The discussion starts for each product area with the most interesting observations from the interviews. These observations are the most important findings that seem to have largest effect on the choice of approach. This addresses **RQ3** by discussing how differences of the product areas and their development processes are affecting the respective approaches.*

*After the observations, some recommendations for each product area are presented. These also addresses **RQ3** but with a future state of the processes in mind. These present how a future choice of approach should be made with concern to the current state of the development process, challenges and advantages of each product area.*

Recommendations may be directly applicable to the product area settings or be presented as general learnings from the observations.

6.1 Product Area A: Observations

Observation A1: The documentation generated from the system model is sometimes hard to interpret.

The system model is used and created with other main purposes than delivering the best documentation. The modelling is done by system engineers, mainly to design and plan all aspects of a system. In the development process, the system model itself is not used by software developers. Only the blueprint of the model is used during implementation. It has shown to be limiting to present models in documented form at product area A. The document is a static artifact, when the real model is very central for tasks and continuously changes. This ends up in that the documentation of the model sometimes hard to interpret or has unclear information.

Observation A2: A lot of software implementation work is refactoring, changing and updating already implemented code. Changes appearing in new system designs causes rework for software developers.

Working with agile approaches, the idea is to be able to respond to change over following a plan. This can mean that incremental work carries some rework with it, related to slightly changed plans or changing circumstances. In the development process at product area A there seems to room for improvement in the different groups' insights into each other's plans. The systems engineers are not aware of the actual implementation at the time and the software developers are surprised by changes in the system design.

Observation A3: The software team works with simplifying maintenance by writing descriptions, painting diagrams and describing flows.

The software developers experience a need to describe the software after the implementation. The developers are to some extent modelling the system by describing

the actual code by diagrams and flows, which is a typical UML modelling task. It is a task done to help understanding code when looking into it later.

Observation A4: Product area A projects are longer, have more teams involved, and concern larger and more complex products than product area B projects.

There are more involved teams and people at product area A, where specialists and team members are from a wider range of engineering disciplines. The system engineers work with complex tasks creating a system design and communicating it to all concerned groups. The projects can be years longer than in product area B and the system has more interfaces towards other systems compared to product area B. Product area B projects are shorter and their products clearly are based on similar solutions. For new projects in product area B, the needed functionality can be mapped to the existing software resources. Software for the product can be created by picking and choosing from modules and changing modules in one or two software layers.

In product area A, the complexity of the products limits the flexibility for making changes to any method or process. The complexity of change projects increases when the product complexity and number of involved teams increase.

Observation A5: Other projects concerning reinventing processes and methods by making software delivery easier through automated release activities are worked with at the time. A platform team is working with reuse methods and common software architecture.

The product area A has a clear direction of projects and investments, where the goal is to have the most competitive software development process possible. The platform team is shaping the future of maintainability, reusability and shared code resources. The automation architects are working with delivery and tasks that concerns the release of software. Both teams are helping the software organization of product area A take further steps towards their ultimate goal, by having teams working full time with change projects side to side with product development teams. It is clear that investments are made to shape the software development processes and make software development easier, quicker and better.

Observation A6: Source code documentation except from comments and explanations of methods and parameters within source code files are not needed in product area A. The synchronization of external documentation and implementation is not described as a motivation for change or a challenge.

Keeping the implementation synchronized with documentation can be hard, because there are two sources to update when code is changed. Product area A has no need to deliver implementation documentation for external purposes, why this is not a problem. They do not struggle with synchronizing these artifacts.

6.1.1 Recommendations

Observation A1: The documentation generated from the system model is sometimes hard to interpret.

Recommendation A1: The documentation could be improved by evaluating how information from the system model could be displayed better. This can be done by investigating most frequently asked questions about the function block design description. It should not be limited to the specific function block the team works with. If the team often needs information from other function blocks or struggles with understanding the system as a whole, this also needs to be included in how to make this work easier. A better documentation of the system model can be internally managed by a team from software development that summarizes the most challenging part of their interpretation work, to start a discussion on how to generate better documentation.

Another solution is that the software developers work with the same artifact as the system engineers, which is the system model. This could help spreading the advantages of modelling to all groups, including communication, visualization and understanding. Practically, it could mean that software developers use the system model to look at it and use different views to understand the system. All software teams can have access to an updated artifact and use the same source of information as all other teams.

Making interpretation of the system model easier could also mean that the system model is to be extended with software models. This implies that there is never a delivery of documentation on the system model. Instead, the model itself is built upon by software developers with UML models of the software. From the models, code can be generated.

Any of the proposed approaches will imply that system engineers must change their methods to ensure better handovers to software developers. Some kind of delivery of models could be necessary.

Observation A2: A lot of software implementation work is refactoring, changing and updating already implemented code. Changes appearing in new system designs causes rework for software developers.

Recommendation A2: What could be improved in the development process are the groups and teams' insight into each other's plans. If system engineers knew and understood the status of the implementations at every moment, it is imaginable that they would be even more observant with how their changes affected or fit the actual implementation and the amount of rework needed. A tighter connected and integrated work would mean that software developers could share their expertise with the system engineers and help with software design challenges.

By implementing software modelling with UML, the system engineers and software developers' work would be more alike than earlier. This could simplify the insight into work of the other group and make the incremental work of both groups coordinated

smoother. If software modelling solves challenges with rework of software, code generation can be easily applied to this as a method for development.

Observation A3: The software team works with simplifying maintenance by writing descriptions, painting diagrams and describing flows.

Recommendation A3: These maintenance enabling tasks would probably not be necessary to the same extent if software was modelled. If software is modelled, a lot of the visualizations will already exist and make software understandable and easily interpretable. Software models can have views describing the exact aspect of interest at a given time. This can enable quick understanding during both development and at maintenance tasks.

Observation A4: Product area A projects are longer, have more teams involved, and concern larger and more complex products than product area B projects.

Recommendation A4: This affect the chosen code generation approach the most. Enabling a full code generation approach for all involved teams and people at product area A is probably not possible to start with. There are so many people involved in developing the products that implementing a full and all through code generation approach would require too much time and effort. A full code generation approach would slow down the development speed and require such extensive learning time it wouldn't be possible to keep up with project plans in any real project setting. Because of the complexity of the product, there is a lot of different software with different purposes. Finding a full-scale code generation approach that fits all teams and all types of software and development is a too complex task.

What can be recommended is gradually moving towards modelling and code generation approaches. Regarding to the project scope, product complexity and number of teams involved, the new approach must both be lead, planned and evaluated. Product area A can gather a team that focuses on modelling and code generation. The team can be a new part of the already existing platform team and/or be made of platform team members. It can also be a completely new team, but must be working closely to the platform team to ensure these investments are in line with each other. The team should consist of both system engineers and software developers, because of the earlier discussed advantages of synchronizing the work of these better. Resources should be available for both these parties to change their internal processes and work to make work better for everyone. All tasks related to controlling, managing and planning the new approach is governed by this team, which ensures dedication to the new approach.

Some software developers or some teams can learn how to model software and generate modules or system components to start with. A special team can be gathered to be test pilots, working with a real project setting but trying to model and code generate instead of coding in the usual IDE. Based on how this works, feedback and learnings can help tailoring the approach to a larger setting internally, making it scalable. The small team

can try both teamwork and agile approaches but also different modelling tools, code generators and technical solutions.

Observation A5: Other projects concerning reinventing processes and methods, software delivery, release tasks, reuse methods and common software architecture are already launched and are worked on by several teams at the time.

Recommendation A5: A successful code generation approach must be in line with the direction of other projects and investments, as well as their expected future outcomes. A modelling or code generation approach should be an enabling and not a disabling factor for these projects. If the approach can be integrated in ongoing investments and in the re-invention of how projects are run within product area A, it will be easier to motivate the change process as well as get internal financial support. The platform team working with reinventing reusability and create a software platform could be a key to smoothly inserting the idea of code generation into the daily practice, as well as making investments go hand in hand.

Practically, this means that some of the reinvented and reusable system components can be the parts that are produced in model form and code generated. Regarding to what should be modelled and how it should be done, some choices need to be made internally by system and software experts that are familiar with the system components. They need to evaluate the reusability of these and the possibility to model and code generate from them. Because of the system complexity, it is not recommended trying to model and code generate all code for the system. The choices are instead either generating a skeleton for the whole product, so the developers has the basic design to start with. Or some system components can be generated and be linked to the rest of the software.

Developers have described the will to have code ready for reusable software components. If the software components are reusable, it means that the code should not be altered or changed for these software components in the different places they are used. Having one model (which can consist of smaller model units) as a source could enable this kind of reuse. Generating code from this model and allowing changes in the source code could make reusable components diverge, which is not preferable. Round-trip engineering will make the management and control of development artifacts harder, which is already a complex task. Full code generation of system components or a product skeleton is the customization of the approach that suits product area A the best.

Observation A6: Source code documentation except from comments and descriptions within source code files are not needed in product area A. The synchronization of documentation and implementation is not described as a motivation for change.

Recommendation A6: The code generation approach should instead focus synchronizing system work with software work and encouraging communication between teams and team members.

Other recommendations:

- Having access to someone experienced in code generation, like hiring a consultant, is important in the change process. This can also be solved by inviting experts from other Saab sites or planning study visits to other Saab sites where modelling and code generation is used. Having someone experienced around during the change process is a recommendation from product area B.

6.2 Product Area B: Observations

Observation B1: Version control tools for models does not facilitate teamwork as intended and forces manual solutions to be used.

Manual work with version control solutions may be a source to less effective and time-consuming tasks. The product area B have also had merging problems because of human factors when developers have forgotten to announce what parts of a model they are working with. A new approach for merging models could make it possible for developers to work at the same parts of the model at the same time. Getting rid of manual announcement solutions could make version control less error prone.

Observation B2: Full code generation approach works well within product area B software developers, but the teams are small and the projects are limited scope.

Product area B have managed to teach all software developers modelling, implement an all through reusable software module platform and work fully with this software platform. It is a method used for all software development tasks and software originating from the software models are in almost all products released since code generation was introduced.

Observation B3: A code generation approach is based on software modelling approaches. Modelling approaches require extensive planning.

Picking a code generator was not a dilemma within product area B. They evaluated the code generator in the modelling environment and thought it was too expensive and that the generated code was not always readable. The main questions regarding to implementing code generation had to do with software modelling and what approach to choose as well as preparations needed.

Observation B4: Both system engineers and software modelers want to extend the software modelling approach with a system modelling approach.

System engineers and software developers within product area B do not work much together. It has been described as a detective's work for system engineers to trace

functionality and they don't see or use the software models for any purpose. The system engineer that has tried modelling on his own hand is very happy with the communicational advantages and is keen on working together with software developers and their models.

Observation B5: Requirements on delivering documentation may be a driving factor for choosing a full code generation approach.

Software developers within product area B are required to deliver documentation on implemented software and are governed by safety standards throughout work. Having synchronized the implementation and documentation through the chosen approach of code generation simplifies documentation work. It helps the team with always ensuring they deliver documentation on the actual implementation.

Observation B6: The chosen tool makes product area B dependent on another Saab site, but it works well for their purposes.

The software developers have described that the code generator is cheap and clearly matches their requirements on a tool because it is imported from another Saab site. They don't alter the code generator even though there are known bugs, and they are dependent on the competence of developers in the other Saab site.

6.2.1 Recommendations

Observation B1: Version control tools for models does not facilitate teamwork as intended and forces manual solutions to be used.

Recommendation B1: Changing the version control solution is one of the recommendations that could mean less errors in merging artifacts. If the modelling and code generation approach is supposed to be scalable to more projects and more team members in the future, the version control solution within product area B needs to be changed. Nor it may in the current state be applicable to larger contexts of embedded systems development. Either a new tool that allows merging models could be used, which would mean that two developers can work on the same part of a model at the same time. Otherwise, a better support for announcing or locking parts of the model is needed. It needs to be impossible for a developer to start working on a model part that is worked on at the same time by someone else.

Observation B2: Full code generation approach works well within product area B software developers, but the teams are small and the projects are limited scope.

Recommendation B2: Choosing a full code generation approach for all software in a product and making it successful is probably not applicable to all projects and teams in the embedded systems software development domain. Totally re-inventing software development where the contexts are more complex would probably require unreasonable loads of preparations and resources.

Observation B3: A code generation approach is based on software modelling approaches. Modelling approaches require extensive planning.

Recommendation B3: The main questions regarding to implementing code generation should be focused on how to do software modelling, to what extent and with what method. What should be modelled? Who should do it? What approach do we choose? What preparations are needed? All aspects related to modelling approaches must be qualitatively evaluated and customized to the context. The code generator could be picked with more quantitative measures like cost, readability and code quality.

Observation B4: Both system engineers and software modelers want to extend the software modelling approach with a system modelling approach.

Recommendation B4: This indicates that a modelling approach isolated to a specific group is not enough to get all the possible advantages of modelling. When working with the same system, in the same project but in different groups and with different tasks, the artifacts central for work should be the same, not varying among groups. If an artifact or an approach isolated to a certain group, all its benefits won't spread.

Implementing an approach that makes it easier to know where functions are allocated and trace system capabilities could improve system work at product area B. It is today described as a detective work to solve bugs and investigate problems. Implementing a model-based systems engineering approach like in product area A could give advantages to planning and designing the system. The products that product area B are working with have been described as based on similar functionality. This could enable the same kind of reuse of system models in the same way that software models have, with a layer methodology of reusable modules.

Today, system engineers have no contact with the software models. This would be changed if the modelling paradigms were linked. There need to be clear ideas upon how models can be connected and how models are delivered or sent between teams. Product area B would see the greatest advantages if they customized their system engineering approach to fit to their code generation and software development approach. At first, this means just starting to work model-based with system design. When the system and software work is linked, it can promote the collaboration between the system and software groups. It would solve traceability problems and give communicational advantages to the whole systems engineering domain.

A team should be formed consisting of both system and software engineers to make sure the two divisions can be linked and have the same idea of how collaboration should be performed. This would promote dialogue between different expertise and shared knowledge in different disciplines.

Observation B5: Requirements on delivering documentation may be a driving factor for choosing a full code generation approach.

Recommendation B5: Teams or organizations not concerned with heavy documentation may not have the same advantage of full code generation.

Observation B6: The chosen tool makes product area B dependent on another Saab site, but it works well for their purposes.

Recommendation B6: Being less or not at all dependent on other sites could make the solution more feasible for future use. It would also mean creating or collecting knowledge on code generators internally instead of having it externally.

This also shows that there can be internal resources or knowledge on code generation already existing in an organization. For all change tasks, there can be clues, resources or help available internally.

Other recommendations:

- Evaluate how code generation be can enabled for components with different standards. Product area B need to think about how to reuse components that are different RTCA levels. Either creating modules that are reusable for the most frequent standard is an idea or creating duplicates of the same module, but with different standards, can be a solution.

7. Conclusions

In this work, the software development processes of two product areas at Saab have been investigated. A mapping of two software development processes has been made to evaluate the current methods and make recommendations for moving towards code generation.

A literature study was made to cover areas of concern, including software development, software modelling, and code generation. The literature study helped to form relevant themes for interviews at the company. Nine interviews were held with system engineers and software developers within the two product areas.

Product area A works with model-based software engineering and uses blueprints of the system design to write code. It is observed that there are challenges with the rework for both system engineers and software developers because of two separate incremental and agile approaches. Recommendations for product area A is to reduce rework by forcing collaboration and in that way, improve insights into the work of other groups. It is discussed that this can be achieved by combining the system model with software models. Software models are not used today within product area A for any task related to design or code generation. It is observed that the projects and products are large-scale and complex, which leads to a recommendation to limit the possible introduction of software modelling to code generation of system components or a system skeleton.

Product area B works with document-based system engineering but a full code generation approach for all software. It is observed that they experience version control problems for software models. The version control solution is recommended to be improved since it leads to problems because of human factors and developers forgetting to announce what model parts they are working on. It is observed how the speed at which problems can be solved can be improved. For product area B projects, it is recommended to focus on tracing functionality and being able to dive deeper from plans into implementation. It is discussed how introducing model-based systems engineering and linking these models to software models could solve traceability issues for system engineers and promote collaboration between groups.

8. Future Research

This thesis has been focusing on modelling and code generation for embedded systems, with large-scale and agile methods and processes. The results, which are in qualitative form, could be a preparation for quantitative studies on the same theme. Some of the presented sub-themes could be investigated with quantitative measures.

- Different tools for software modelling could be investigated by comparing the time it takes to model relevant embedded software.
- Different code generators could be used to generate code from the same model. The resulting code from different generators could be compared regarding code generation time, code complexity, errors and other measures.
- Pipeline data from the product area B development processes before and after code generation was introduced could be investigated. Time to release, the amount of time needed to solve similar tasks could be compared to prove productivity gains.
- The quality of produced code could be compared between product area A and product area B. It could give clues to what approach is less error-prone and described in terms of bugs per code row or similar measures.

Furthermore, there are openings for diving deeper into some of the presented solutions and approaches. An example would be focusing on how teamwork with models diverge from teamwork with code, how work is shared and planned. Investigating how version control solutions with models affect teamwork and possible solutions is another theme related to teamwork.

The context of the study could be slightly changed to see how other different settings related to embedded systems development affect the approach chosen. Evaluating how different settings, contexts, and methods diverge within the embedded systems domain could give even more clues to how modelling approaches vary, and why they do. Investigating a lot of different contexts in the domain could give a broader understanding of how to customize approaches and methods to any development setting.

References

Ali, N.B., Petersen, K., Schneider, K. (2016), "FLOW-assisted value stream mapping in the early phases of large-scale software development", The Journal of Systems and Software, vol. 111, pp. 213-227. [Available online: http://bth.diva-portal.org/smash/get/diva2:881321/FULLTEXT01.pdf (2018-01-16)]

Anda, B., Hansen, K., Gullesen, I., Thorsen, H.K. (2006), "Experiences from introducing UML-based development in a large safety-critical project", Empirical Software Engineering, vol. 11, no. 4, pp. 555-581. [Available online: https://link-springer-com.ezproxy.its.uu.se/content/pdf/10.1007%2Fs10664-006-9020-6.pdf (2018-01-18)]

Bassi, L., Secchi, C., Bonfé, M., Fantuzzi, C. (2011), "A SysML-Based Methodology for Manufacturing Machinery Modelling and Design", IEEE/ASME Transaction on Mechatronics, vol. 16, no. 6, pp. 1049-1062. [Available online: https://ieeexplore-ieee-org.ezproxy.its.uu.se/stamp/stamp.jsp?tp=&arnumber=5604318 (2017-12-07)]

Biju, S. M. (2008), "Agile Software Development", E-Learning and Digital Media, vol. 5, no. 1, pp. 97-102. [Available online: http://journals.sagepub.com.ezproxy.its.uu.se/doi/pdf/10.2304/elea.2008.5.1.97 (2018-01-17)]

Brambilla, M., Cabot, J., Wimmer, M. (2017), Model-driven software engineering in practice, 2nd Edition. Morgan & Claypool: San Francisco, CA. [Available online: https://www-morganclaypool-com.ezproxy.its.uu.se/doi/pdf/10.2200/S00751ED2V01Y201701SWE004 (2018-01-17)]

Cervone, H.F. (2011), "Understanding agile project management methods using Scrum", OCLC Systems & Services: International digital library perspectives, vol. 27, no. 1, pp. 18-22. [Available online: https://www-emeraldinsight-com.ezproxy.its.uu.se/doi/pdfplus/10.1108/10650751111106528 (2018-02-01)]

Chaudron, M.R.V. (2017), "Empirical Studies into UML in Practice: Pitfalls and Prospects", IEEE/ACM 9th International Workshop on Modelling in Software Engineering, pp. 3-4. [Available online: http://www.di.univaq.it/diruscio/sites/mise2017/Michel-Chaudron-MISE17-Invited.pdf (2018-02-19)]

Chaudron, M.R.V., Heijstek, W., Nugroho, A. (2012), "How effective is UML modelling ?", Software and Systems Modelling, vol. 11, no. 4, pp. 571-580. [Available online: https://link-springer-com.ezproxy.its.uu.se/content/pdf/10.1007%2Fs10270-012-0278-4.pdf (2018-02-02)]

Cicchetti, A., Ciccozzi, F., Carlson, J. (2016), "Software Evolution Management: Industrial Practices", ME@MODELS. [Available online:

https://pdfs.semanticscholar.org/4eda/277813d611ad3f472d0ddb4b5d1f354d1ad4.pdf?_ga=2.52609963.2005621690.1524059134-564666837.1523620010 (2018-01-23)]

Clarke, P., O'Connor, R.V. (2012), "The situational factors that affect the software development process: Towards a comprehensive reference framework", Information and Software Technology, vol. 54, no. 5, pp. 433-447. [Available online: https://www-sciencedirect-com.ezproxy.its.uu.se/science/article/pii/S0950584911002369?via%3Dihub (2018-01-20)]

Crnkovic, I., Stafford, J. (2013), "Embedded Systems Software Architecture", Journal of Systems Architecture, vol. 59, no. 10, part D, pp. 1013-1014. [Available online: https://www-sciencedirect-com.ezproxy.its.uu.se/science/article/pii/S1383762113002415?via%3Dihub (2018-02-01)]

Ejvegård, R. (2009), Vetenskaplig metod, 4th Edition. Studentlitteratur: Lund.

Eriksson, L.T., Wiedersheim-Paul, F. (2011), Att utreda, forska och rapportera, 9th Edition. Liber: Malmö.

Franky, M. C., Pavlich-Mariscal, J. A., Acero, M. C., Zambrano, A., Olarte, J. C., Camargo, J., Pinzón, N. (2016), "ISML-MDE: A practical experience of implementing a model-driven environment in a software development organization", International Journal of Web Information Systems, vol. 12, no. 4, pp. 533-556. [Available online: https://doi.org/10.1108/IJWIS-04-2016-0025 (2017-12-07)]

Glass, R.L. (2003), Facts and fallacies of software engineering. Addison-Wesley: Boston, MA.

Greer, D., Hamon, Y. (2011), "Agile Software Development", Software: Practice and Experience, vol. 41, no. 9, pp. 943-944. [Available online: https://onlinelibrary-wiley-com.ezproxy.its.uu.se/doi/epdf/10.1002/spe.1100 (2018-01-16)]

Grösser, S., Reyes-Lecuona, A., Granholm, G. (2017), "Dynamics of Long-Life Assets", pp. 169-189, Springer: Cham. [Available online: https://link.springer.com/book/10.1007%2F978-3-319-45438-2#toc (2018-04-23)]

Heldal, R., Pelliccione, P., Eliasson, U., Lantz, J., Derehag, J., Whittle, J. (2016), "Descriptive vs Prescriptive Models in Industry", ACM, pp. 216-226. [Available online: https://dl-acm-org.ezproxy.its.uu.se/citation.cfm?id=2976808 (2018-03-02)]

INCOSE (2011), Systems Engineering Handbook, International Council on Systems Engineering, version 3.2.2. INCOSE: San Diego.

Jörges, S. (2013), Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach. Springer: New York, Berlin.

Labrosse, J.J. (2007), Embedded Software. Newnes: Oxford.

Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J. (2014), "Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain", 17th International Conference on Model-Driven Engineering Languages and Systems, vol. 8767, pp. 166.

MacCormack, A., Verganti, R. (2003), "Managing the Sources of Uncertainty: Matching Process and Context in Software Development", Journal of Product Innovation Management, vol. 20, no. 3, pp. 217-232. [Available online: https://onlinelibrary-wiley-com.ezproxy.its.uu.se/doi/abs/10.1111/1540-5885.2003004 (2018-02-08)]

Marincic, J., Mader, H. A., Wieringa, R., and Lucas, Y. (2013), “Reusing knowledge in embedded system modelling”, Expert Systems, vol. 30, no. 3, pp. 185-99. [Available online: https://onlinelibrary-wiley-com.ezproxy.its.uu.se/doi/abs/10.1111/j.1468-0394.2012.00631.x (2018-04-27)]

Mattsson, A., Fitzgerald, B., Lundell, B., Lings, B. (2009), "Linking Model-Driven Development and Software Architecture: A Case Study", IEEE Transactions on Software Engineering, vol. 35, no. 1, pp. 83-93. [Available online: https://ieeexplore-ieee-org.ezproxy.its.uu.se/stamp/stamp.jsp?tp=&arnumber=4657364 (2018-01-16)]

Mattsson, A., Fitzgerald, B., Lundell, B., Lings, B. (2012), "An Approach for Modelling Architectural Design Rules in UML and its Application to Embedded Software", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 21, no. 2, pp. 1-29. [Available online: https://dl-acm-org.ezproxy.its.uu.se/citation.cfm?id=2089120 (2018-01-22)]

Object Management Group (2017), About the OMG System Modelling Language Specification Version 1.5. [Available online: https://www.omg.org/spec/SysML/1.5/ (2018-04-18)]

Rumpe, B. (2017) Agile Modelling with UML: Code Generation, Testing, Refactoring. Springer: Cham. [Available online: https://link-springer-com.ezproxy.its.uu.se/book/10.1007%2F978-3-319-58862-9 (2018-05-02)]

Selic, B. (2003), "The pragmatics of model-driven development", IEEE Software, vol. 20, no. 5, pp. 19-25. [Available online: https://ieeexplore-ieee-org.ezproxy.its.uu.se/stamp/stamp.jsp?tp=&arnumber=1231146 (2018-02-02)]

Trendowicz, A. (2013), Software Cost Estimation, Benchmarking, and Risk Assessment. Springer: Heidelberg. [Available online: https://link-springer-com.ezproxy.its.uu.se/content/pdf/10.1007%2F978-3-642-30764-5.pdf (2018-01-16)]

White, E. (2011), Making Embedded Systems. O'Reilly: Sebastopol, CA.

Whittle, J., Hutchinson, J., Rouncefield, M. (2014), "The State of Practice in Model-Driven Engineering", IEEE Software, vol. 31, no. 3, pp. 79-85. [Available online: https://ieeexplore-ieee-org.ezproxy.its.uu.se/stamp/stamp.jsp?tp=&arnumber=6507223 (2018-01-18)]

Zhang, Y., Patel, S. (2011), "Agile Model-Driven Development in Practice", IEEE Software, vol. 28, no. 2, pp. 84-91. [Available online: https://ieeexplore-ieee-org.ezproxy.its.uu.se/stamp/stamp.jsp?tp=&arnumber=5456357 (2017-12-07)]

Appendix

Interview Protocol – Standard questions

A.1 Introduction

- Introduction of writer, supervisor and aim of project
- Sound recording or taking notes, discussion
- Agree on time limit
- Confidentiality or secrecy discussion

A.2 Background of Interviewee

- What is your official job title?
- What are your assigned tasks and responsibilities?
- How long have you been at this role and what is your background in the organization?
- What is your background in modelling and/or code generation?

Insert questions from Process Mapping (PM.1-8)

A.3 Summary

- Do you have any questions to me?
- Can you propose 1-3 people that you think I should interview?
    - What are their roles and what can they describe more detailed?
- Show work process map and ask for feedback
- Ask if the interviewee wants to read through the transcriptions of their interview
- Thank the interviewee for the time

Interview Protocol - Process Mapping

PM.1 Projects

- What project(s) are you currently working on?
    - Can you give a general overview of the project aims?
- What are your tasks in the project?
    - Method
    - Tools
    - Key persons in your role

PM.2 Systems Engineering/Development

- What part of Systems Engineering does your work cover or interact with?
- What key persons in Systems Engineering are you dependent on or are dependent on you?
- What part of Development does your work cover or interact with?
- What key persons in Development are you dependent on or are dependent on you?

PM.3 Models

- Are you using model as an artifact in your daily work?

If **no**: PM.4

If **yes**:

  - For what purposes do you use modelling?
  - What modelling languages?

- How do you use models in the projects you have been participating in?
- Do you have an opinion on models in system and software development?
  - What are the biggest advantages of modelling?
  - What are the biggest disadvantages or challenges of modelling?
- What models do you use and not use? How are they related or connected?
- How are models tested or validated?
- How does traceability work back to requirements and/or to code?
- How is a model changed or altered during its lifetime and why?
- How is a model version controlled? (If it is)
- Are you using code generation?

If **no**: PM.4

If **yes**: CG.1 (Interview Protocol B)

PM.4 Communication

- How are decisions made?
- What communication (meetings, emails, face to face) do you use in your work?
- How do you think that communication between project members is going?

PM.5 Documentation

- What information from documents do you use in your work?
- What information in form of documents do you create in your work?
- How do you feel about documentation as it works right now?

PM.6 Development

- How does delivery of artifacts from systems engineering to development work?
- How do developers work with artifacts from systems engineering?
- How are artifacts used in the development process?
- How is plan, design and implementation kept in line with each other?
- How do you work with maintenance and enabling maintenance?
- Do you have any external safety standards to follow?
- What feedback comes from development to systems engineering?
  - What kind of communication?
  - To what key persons?

PM.7 Process

- Describe the way of working (like agile or waterfall)?
- Describe what happens when software is released?

PM.8 Thoughts

- What have you heard of code generation?
- What is your general thought of code generation?
- Do you have any ideas on how it should be done?

Interview Protocol B – Code Generation

CG.1 Approach choice

- Why and when was code generation introduced?
- Did anyone have experience of code generation when it was introduced?
- What was the general thought of code generation when it was introduced?
- Who does code generation?
    - What role and expertise does he/she have?

CG.2 Tools

- What tool is used for code generation?
- Why was this specific chosen?
    - How was reliability discussed in the choice?
    - How was safety-critical software discussed in the choice?
- How has the tool been altered, changed or adjusted?
- Does code generation cover all aspects of development you want it to cover?

CG.3 Related to modelling

- How has code generation affected modelling efforts?
- Are artifacts being reused in any way?
- How are models and code synchronized?
- Are design decisions from models easier to interpret with code generation?

CG.4 Processes

- How did code generation match with existing processes when it was implemented?
- How do you work with enabling easy maintenance?
- How has code generation effected cycle time or process time?

CG.5 Changes with code generation

- How has code generation brought training time and costs?
- What do developers think of code generation today?
- How is the quality of generated code (bugs and errors)?
- How has code generation changed communication between people and roles?