Studying the benefits of using UML on software maintenance : an evidence-based approach

Fernandez Saez, A.M.

CHAPTER 6. WHAT IS THE IMPACT OF USING UML IN SOFTWARE MAINTENANCE IN TERMS OF ITS USE IN A SOFTWARE PROJECT?

Fernández-Sáez, A. M., Chaudron, M. R. V., and Genero, M. (2018). An industrial case study on UML modelling practices and their effectiveness in software maintenance. *Empirical Software Engineering* (**JCR Index 2016 = 3.275;** published online 16th March 2018).

ABSTRACT

Context: UML is a commonly-used graphical language for the modelling of software. Works regarding UML's effectiveness have studied projects that develop software systems from scratch. Yet the maintenance of software consumes a large share of the overall time and effort required to develop software systems. In this study we, therefore, focus on the use of UML in software maintenance.

Objective: Our goal is to elicit the practices of the software modelling used during maintenance in industry. Moreover, we wish to understand what are perceived as hurdles and benefits when using modelling.

Method: In order to achieve a high level of realism, we performed a case study in the ICT department of a multinational company. The analysis is based on 31 interviews with employees who work on software maintenance projects. The interviewees played different roles and provided complementary views about the use, hurdles and benefits of software modelling and the use of UML.

Results: Our study uncovered a broad range of modelling-related practices, which are presented in a theoretical framework that illustrates how these practices are linked to the specific goals and context of software engineering projects. We present a list of recommended practices that contribute to the increased effectiveness of software modelling.

Conclusions: The use of software modelling notations (like UML), is considered beneficial for software maintenance, but needs to be tailored to its context. Various practices that contribute to the effective use of modelling are commonly overlooked. This suggests that a more conscious and holistic approach with which to integrate modelling practices into the overall software engineering approach is required.

Keywords: UML, Software Maintenance, Modelling Languages, Industrial Case Study

6.1. INTRODUCTION

Modelling is a common practice in software engineering, and UML (OMG, 2011) is the de-facto standard notation for this (Dobing and Parsons, 2006; Scanniello et al., 2010). However, in-depth evidence regarding the practice of using UML in industry is very scarce. Studies into the effectiveness of UML have looked mostly at projects that develop software systems from scratch. Yet the maintenance of software consumes a large share of the overall time and effort required to develop software systems. Unlike developers, software maintainers, spend a lot of their time understanding the system. This has led us to pay particular attention to the use of UML diagrams by the maintainers of software systems, which consequently motivated this study. We believe that industrial studies of this nature on software modelling could contribute to an understanding of how modelling can be used effectively. Empirical studies are necessary in real environments if we are to increase the depth of the knowledge and validity of the application of Software Engineering practices (Runeson and Höst, 2009).

A considerable amount of software development effort concerns the maintenance of software. Indeed, maintenance typically consumes between 40% to 80% of software project costs (Pressman, 2005). It is, therefore, important to understand the impact that software modelling may have on software maintenance. We are convinced that the way in which UML is used – i.e. which practices are used− is an important contextual factor that is associated with the effectiveness of the use of UML. In order to understand the effectiveness of UML, we have elicited factors related to the costs and benefits of using modelling during software maintenance. In doing so, we add fresh findings to the hitherto scarce evidence regarding the payoffs and costs of software modelling.

There is only scant empirical evidence related to our topic, the majority of which has been obtained in academic contexts (Dzidek et al., 2008; Fernández-Sáez et al., 2014, 2015a). However, it is important to investigate whether the results obtained in a controlled context are valid in an industrial one. A case study explores a phenomenon within its real context, especially when the boundaries between phenomenon and context are not evident (Yin, 2002). The essence of the case study method is to conduct an empirical inquiry within its real-life context and to thereby provide detailed, qualitatively rich, contextual description and analysis of a complex real-life phenomenon. The relevance of the findings should be sought in not only their novelty, but also in the confirmation of 'known' practices, as well as in the contextualisation of the object of study. We therefore believe that the case study approach is suitable for our study, because we wish to gather detailed information concerning an industrial software maintenance project and to obtain reflections from practitioners as regards the failures and success (especially in terms of costs and benefits) involved in the use of UML diagrams (or in their absence). In this paper, then, we present empirical findings obtained in the ICT (Information and Communication Technology) department of a large multinational company by means of case study research.

The principal goal of our research is to discover what practices industrial software professionals employ when using UML, and how they perceive the effectiveness of software modelling, paying particular attention to software maintenance tasks. As mentioned above, we focus our attention particularly on UML as a specific modelling language because it is widely used in industry (Dobing and Parsons, 2006; Scanniello et al., 2010).

Since the term 'maintenance' is very broad, we would like to limit its scope from the beginning of this paper. That being so, in this paper, "maintenance" always refers to adaptive, perfective and corrective maintenance (Pigoski, 2001), while preventive maintenance is out of the scope of this study.

This paper is organised as follows. Section 6.2 presents the related work. Section 6.3 describes the case study and its design. The results obtained are set out in Section 6.4, whilst some recommendations are provided in Section 6.5. In Section 6.6 a summary of all the results, organised by research question, is presented, while the

threats to validity are discussed in Section 6.7. Finally, Section 6.8 outlines our main conclusions, and Section 6.9 presents future work.

6.2. RELATED WORK

In order to find the relevant work related to this paper, we performed a Systematic Mapping Study (SMS) to discover all the empirical studies related to the use of UML diagrams in software maintenance, and to the understandability and modifiability of UML diagrams as regards how they might influence system maintenance (Fernández-Sáez et al., 2013a). The findings obtained in this SMS are presented in the first block of related work. As the SMS was performed a few years ago, we have attempted to update its results to discover more recent evidence related to the topic discussed in this paper. In the second part of this section, we go on to present experiments or academic evidence related to the use of UML. The last block of this section is similar to the second, because it also complements the evidence found in the SMS, but it contains industrial rather than academic evidence.

The SMS (Fernández-Sáez et al., 2013a) enabled us to discover 46 papers reporting 74 empirical studies related to the topic of the use of UML diagrams in software maintenance. Of these, only the following experiments were directly related to the maintenance of source code:

- Dzidek et al. (Dzidek et al., 2008) performed a controlled experiment to investigate whether the use of UML influences maintenance, as compared to the use of only source code. This experiment investigates the costs of maintaining, along with the benefits of using, UML documentation during the maintenance and evolution of a real nontrivial system, with 20 professional developers as subjects. These maintainers had to perform five maintenance tasks consisting of adding new functionalities to an existing system, after which the correctness, time and quality of the solution were measured. Both the source code and the UML diagrams, when available, had to be maintained. The results of this work show a positive influence of the presence of UML for maintainers. UML was always beneficial in terms of functional correctness (with fewer faults being incorporated into the software) because the subjects in the UML group showed, on average, a practically and statistically significant 54 percent increase in the functional correctness of changes. UML also helped produce code of better quality when the developers were not yet familiar with the system. This experiment is a replication of previous work (Arisholm et al., 2006) performed with students, which obtained similar results.
- Arisholm et al. (Arisholm et al., 2006) presented the results of two controlled experiments carried out to assess the impact of UML design diagrams on software maintenance. 98 undergraduate students were involved. The authors analysed the time taken to perform the modifications to the system, the time spent on maintaining the models, and the quality of the modifications performed. The results of the quantitative analysis revealed no significant difference as regards the time spent making the modifications. Like (Dzidek et al., 2008), they observed that the quality of the modifications was higher for those participants who were equipped

with UML diagrams. As in (Dzidek et al., 2008), the participants' ability and experience were not analysed with regard to the comprehensibility and modifiability of source code. Unlike our study, the authors analysed the effect of UML-based documentation (a use case diagram, sequence diagrams for each use case, and a class diagram) on modification tasks performed on both UML diagrams and source code.

In order to extend the results obtained in the aforementioned SMS, we would like to mention some other empirical studies, which were not part of the SMS, owing to their particular contexts or dates. We shall first briefly describe other recent experiments performed using undergraduate students, and which are related to UML and maintenance:

- A comparison of the attitude and performance of maintainers when using forward designed (FD) diagrams vs. reverse engineered (RE) diagrams during the maintenance of source code is discussed in (Fernández-Sáez et al., 2015a). The results show a tendency towards obtaining better results when using UML diagrams (class diagrams specifically) that were hand-made during the design phase. This is because the participants preferred FD diagrams when understanding and maintaining a system, although their performance was not, in some cases, much better with FD diagrams. They also noted that those participants who received RE diagrams had more difficulties when reading the diagrams, especially the sequence diagrams.
- Focusing on the possible advantages of Model-Driven Development (MDD), improvement to maintainability is studied in an experiment presented in (Ricca et al., 2012). The results, which were obtained with Unimodal (a specific implementation of executable UML), indicate a relevant shortening of time with no significant impact on correctness, in comparison to conventional manual programming when performing maintenance tasks.
- In (Scanniello et al., 2012) an experiment is presented, whose aim was to assess whether the comprehension of source code is influenced by the presence or otherwise of UML class and sequence diagrams produced in the design phase. The results reveal that the availability of UML allowed the subjects to perform the maintenance tasks better.
- The results of a family of 4 controlled experiments presented in (Scanniello et al., 2014) reveal that the use of analysis-level UML diagrams does not significantly improve the comprehension and modifiability of source code with regard to the use of source code alone.
- The experiment in (Fernández-Sáez et al., 2014) studies whether different Levels of Detail (LoD) in UML diagrams influence the maintenance of source code. The results suggest that high LoD diagrams are more helpful when understanding a system in comparison to low LoD, while low LoD diagrams are more helpful when carrying out maintenance tasks.

- Leotta et al. (Leotta et al., 2013) conducted a pilot experiment with 21 Bachelor's degree students, aiming to investigate the effect of documentation accuracy during software maintenance and evolution activities. The result they obtained was a benefit of 15% in terms of efficiency when more accurate documentation was used. Part of their experiment revealed that UML documentation is considered to be more accurate.

The pattern that emerges from the results of these experiments, under controlled conditions in academic environments, is that the use of UML diagrams is, to some extent, beneficial for software maintenance. One important issue was to study whether these results also hold in an industrial environment. The industrial evidence related to the topic of this paper is the following:

- Scaniello et al. (Scanniello et al., 2010) presented the results of an exploratory survey used to investigate the state of practice regarding the use of UML in software development and maintenance. The majority of the companies interviewed (75% out of 22 companies) used UML for software development and maintenance. The interviewees were mainly practitioners with little experience as regards performing maintenance operations. Another interesting point concerns the average effort needed to perform maintenance operations, which ranges from 1 to 5 person hours for an ordinary maintenance operation (e.g., corrective changes), and from 10 to 50 person hours in the case of an extraordinary maintenance operation (e.g., perfective or adaptive changes). Scaniello's study provides the responses of 22 Italian companies, which means that the study cannot be generalised to other companies throughout the world.
- Several related pieces of work investigate aspects of software modelling in general and/or model-driven approaches (MDA). If we focus on those surveys directly related to the influence of software modelling with software development, the work of Anda et al. (Anda et al., 2006) should be highlighted. The paper in question reported the anecdotal advantages of modelling, such as improved traceability, but also pointed to potential negatives, such as increased time taken to integrate legacy code with models, and organisational changes needed to accommodate modelling.
- There is also a paper based on the study of Model Driven Engineering (MDE) from an empirical point of view (Hutchinson et al., 2014). The authors first present the results of a survey of MDE deployment, and provide some rough quantitative measures of MDE practices in industry. They then go on to supplement these figures with qualitative data obtained from some semi-structured, in-depth interviews with MDE practitioners In particular, they supplement their figures by adding a description of the practices of four commercial organisations as they adopted a MDE approach for their software development practices. In documenting some details of their attempts to deploy model driven practices, the authors identify a number of contextual factors, in particular the importance of complex organisational, managerial, and social factors (as a complement to technical factors) that appear to influence the relative success, or failure, of the endeavour.

The interviewees describe genuine success in their use of model driven development, but explain this as examples of organisational change management. They conclude that the successful deployment of MDE appears to require a progressive and iterative software development approach, transparent organisational commitment and motivation, integration with existing organisational processes, and a clear business focus.

All the related work is summarised in Table 6.1, thus providing the reader with an overview of the main information with which to compare the empirical studies. The columns in the table have the following content:

- **Ref**: contains the reference to the paper that presents the empirical study considered.
- **Type of empirical study**: indicates the type of empirical study summarised in the paper (a survey, an experiment, a family of experiments, etc.).
- **Goal**: describes the goal pursued by the empirical study.
- **Subjects**: presents the numbers of subjects who participated in the empirical studies, as well as the type of subjects (students, professionals, academic staff, etc.).
- **Independent variables**: describes the variables that are studied, to ascertain their effect on the dependent variables. The values (treatments) of the independent variables are also presented.
- **Dependent variables**: presents the outcome variables, which are the variables that are affected by the changes produced in the independent variables.
- **Experiment design**: contains the type of design selected, which can be between-subjects (each subject receives only one treatment) or within subjects (each subject receives all the treatments).
- **Tasks**: describes the tasks to be performed by the subjects as part of the empirical study.
- **Results**: reveals the main findings obtained.

6.3. RESEARCH METHOD

As mentioned above, we selected the case study as the research method that would be applied, in order to obtain empirical evidence from a real environment. We believe the case study approach is suitable for our study because we wish to gather detailed information about industrial software maintenance projects and to obtain reflections from practitioners on the failures and success in relation to the use of UML diagrams (or their absence).

In this section, we discuss aspects of the research method employed in our study, following the guidelines for case studies proposed by (Cruzes et al., 2011; Höst and Runeson, 2007; Runeson and Höst, 2009; Runeson et al., 2012).

Table 6.1. Summary of related work (part 1/3).

Ref	(Dzidek et al., 2008)	(Arisholm et al., 2006)	(Fernández-Sáez et al., 2015a)	(Ricca et al., 2012)
Type of study	1 experiment	2 experiments	Family of experiments (1+2)	1 experiment
Goal	To investigate whether the use of UML influences maintenance in comparison to the use of only source code.	To investigate whether the use of UML influences maintenance, in comparison to the use of only source code.	To determine the influence of the origin of UML diagrams on source code maintenance.	To evaluate the effectiveness of Model-driven development (using UniMod) during software maintenance tasks.
Subjects	20 professional developers.	Undergraduate students (22 and 76, respectively).	Undergraduate students (40, 51 and 78, respectively).	21 Bachelor's degree students.
Independent variables	The use of UML documentation in a UML-supported IDE (possible values: presence or absence of UML diagrams accompanying source code).	The use of UML documentation in a UML-supported IDE (possible values: presence or absence of UML diagrams accompanying source code).	The origin of UML diagrams (possible values: forward design or reverse engineered diagrams).	Effectiveness of UniMod vs. Java.
Dependent variables	-Time needed to change source code. -Time needed to change source code + UML diagrams. -Functional correctness of the solution.	-Time needed to change source code. -Time needed to change source code + UML diagrams. -Functional correctness and quality of the solution. -Quality of the change.	-Understandability of source code. - Modifiability of source code.	- Time required to perform maintenance tasks. - Artefact correctness.
Design	Between-subject design.	Between-subject design.	Between-subjects balanced design.	Counter-balanced design.
Tasks	Modification tasks in source code and in UML diagrams.	Modification tasks in source code and in UML diagrams.	Modification tasks + Subjective questions.	Maintenance tasks in source code and in UniMod.
Results	The UML subjects took more time if the UML documentation was to be updated. UML was always beneficial in terms of functional correctness. UML also helped produce better quality code when the developers were not yet familiar with the system.	The UML subjects took more time if the UML documentation was to be updated. UML was always beneficial in terms of functional correctness. UML also helped produce better quality code when the developers were not yet familiar with the system.	Tendency to obtain better results when using class diagrams, which were hand-made during the design phase based on statistical and subjective results. Reversed-engineered sequence diagrams were considered difficult to read.	Results indicate a relevant shortening of time with no significant impact on correctness, gained through the use of UniMod instead of conventional programming (i.e. code centric programming).

Table 6.1. Summary of related work (part 2/3).

Ref	(Scanniello et al., 2012)	(Scanniello et al., 2014)	(Fernández-Sáez et al., 2014)
Type of study	1 experiment	Family of experiments (1+3)	Family of experiments (1+3)
Goal	To investigate whether the comprehension of source code increases when participants are provided with UML class and sequence diagrams produced in the software design phase.	To investigate whether the use of UML models produced in the requirements analysis process helps in the comprehensibility and modifiability of source code.	To determine the influence of different levels of detail (LoD) of UML diagrams in source code maintenance.
Subjects	16 undergraduate students.	Undergraduate students (24, 22, 22 and 18, respectively).	Undergraduate students (11, 16, 32 and 22, respectively).
Independent variables	The use of sequence and class diagrams created in the design phase (possible values: presence or absence of UML diagrams accompanying source code).	The use of UML analysis models (possible values: presence or absence of UML diagrams accompanying source code).	The level of detail of UML diagrams (possible values: high or low LoD).
Dependent variables	Comprehension of source code.	-Comprehension level of the source code. -Capability of a maintainer to modify source code.	Understandability and modifiability of source code.
Experiment design	Comprehension questions.	Comprehension and modification tasks and subjective questions.	Comprehension tasks + modification tasks + subjective questions.
Tasks	Within-subjects.	Within participants Counter-balanced design.	Between-subjects.
Results	Participants comprehend source code significantly better when class and sequence diagrams are added together.	UML models produced in the requirements analysis process influence neither the comprehensibility of source code nor its modifiability.	The descriptive statistics show that high LoD diagrams are more helpful when understanding a system in comparison to low LoD, while low LoD diagrams are more helpful when carrying out maintenance tasks.

Table 6.1. Summary of related work (part 3/3).

Ref	(Leotta et al., 2013)	(Scanniello et al., 2010)	(Anda et al., 2006)	(Hutchinson et al., 2014)
Type of study		1 survey	1 case study	1 survey + interviews
Goal	To investigate the impact of non-aligned UML documentation on maintenance tasks.	To investigate the state of the practice regarding the use of UML in software development and maintenance in Italian industry.	To identify immediate benefits, along with difficulties and their causes, when introducing UML-based development into large projects.	To describe and understand the industrial experience of MDE.
Subjects	21 third year Bachelor's degree students.	-	16 system developers and project managers.	-
Independent variables	The alignment of class diagram and source code (Possible values: "less"/ "more" aligned documentation)	-	-	-
Dependent variables	Efficiency	-	-	-
Experiment design	Source code maintenance tasks	-	-	-
Tasks	Counter-balanced design.	-	-	-
Results	The results confirmed that an aligned documentation helps during maintenance tasks (15% benefit).	The majority of the companies interviewed use UML for software development and maintenance. Practitioners with little experience perform mainly maintenance perations.	The interviewees made improvements to traceability from requirements to code, design of the code, and development of test cases, and also to communication and documentation.	Some factors that appear to influence the success/failure of applying MDE approaches, as opposed to simple technical factors, were identified: in particular, the importance of complex organizational, managerial and social factors.

6.3.1. Goal and research questions

In the introduction, we affirmed that the principal goal of our research is to discover what practices industrial software professionals use when using UML, as well as to find out how they perceive the effectiveness of software modelling, paying particular attention to software maintenance tasks.

The use of the Goal-Question-Metrics template (Basili and Weiss, 1984) enabled us to formulate the goal of this case study as follows: "***Analyse*** *the use of software modelling* ***for the purpose of*** *investigating its use, effectiveness and hurdles,* ***with regard to*** *software maintenance tasks,* ***from the perspective of*** *the researcher,* ***in the context of*** *an industrial ICT department".*

This goal was used to formulate our main research questions (RQs):

- *RQ1) What practices are involved in using UML in software maintenance projects?*
  This research question was directly derived from the GQM statement. We wanted to understand the impact of UML, and suspected that different people used UML in different ways. Through this question, we wanted to elicit the variety of practices, because different practices are likely to have different kinds of impact on effectiveness of use of UML.
- *RQ2) What are the costs-factors and benefit-factors of using UML in software maintenance projects?*
  This research question arose from RQ1, because costs and benefits are drivers that shape the practices of using UML. This RQ thus needs to be answered in an effort to help find a response for RQ1.

Although the main objective of this study is to attempt to answer the RQs stated above, two additional RQs were also considered when designing and analysing this case study. The following RQs were added, so as to provide a comparison between UML documentation, and documentation that does not contain diagrams (UML diagrams in particular, and other modelling notations in general):

- *RQ3) What are the factual and perceived hurdles when maintaining documentation, and UML models as part of that documentation?*
  Apart from providing a comparison of documentation containing models vs. documentation which does not cointain any kind of models, this RQ separates the subjective point of view concerning the obstacles to the correct maintenance of the documentation of a project found by maintainers from the factual, objective perspective.
- *RQ4) What are best practices when using diagramming and modelling in documentation?*
  After eliciting and reflecting on the costs and benefits, as well as the hurdles experienced by maintainers, we try to distil what can be considered the best practices, and go on to provide some recommendations about (UML) modelling documentation, and SE practices in general.

6.3.2. Case and subject selection

For our case study, we collected data at the ICT Department of a multinational transport company in Western Europe. This company is about 100 years old, but we could trace the use of ICT in the company back to (at least) the late 1960's. The ICT department has between 800-1000 employees, all of whom are involved in ICT functions. ICT plays an essential role in this company's competitive advantage within its sector. Moreover, the company has to comply with many ICT standards in the sector. An internal department produces the company's software in-house, and its innovation as regards ICT is a valuable asset. The organisational unit of interest for this research is located within the Information Services development department, which is part of the ICT department. The Information Services development department is where all the software development takes place.

The types of software systems that are being maintained at the company are the following:

- **Mainframe systems**: these started in the late 60's, signifying that low level programming languages are used. Programmers involved in the maintenance of this kind of systems are hired after completing their school studies, and trained in-house. This training is based on high-ceremony and documentation-intensive development processes.
- **Information systems**: this kind of systems includes several hundreds of applications, several of which are considered to be legacy systems. They usually contain a fair number of interfaces to external systems. Employees with heterogeneous backgrounds perform the maintenance of this kind of systems.
- **Mobile systems**: these systems typically use web and agile technologies. Younger developers who have had some form of UML training as part of their formal education are usually involved in the maintenance of this kind of systems.

The ICT employees work in different divisions. These divisions are organised by domain-area. Another organisational split is in business technology. People in the technology domain are grouped in technology-areas: mainframe, mobile and web, Java, SAP, etc. The ICT organisation can therefore be considered as a matrix structure (Figure 6.1). The functions that people have in a team may additionally be specialised in a particular activity of the software development process: architect, developer, tester, tool-smith, business-analyst, information-analyst, project manager, etc.

In this organisation, all software is developed through projects, which have a limited duration in time. When a new system is provided, the Company call that project a "development project", while an existing system that is modified or adapted is called a "maintenance project" (that is, large maintenance changes are made in the form of new development projects). The creation and update of documentation is part of the responsibility of the projects themselves. Software maintenance is carried out by a specialised and dedicated team in the organisation. Most of the projects in this ICT department are in fact mainly software maintenance in type. We shall therefore

provide a detailed definition of maintenance (which is also the explanation we gave to all the participants in the case study).

	Division 1		Division 2	Division 3		
Domain 1	Java	C#	.NET	Java	SAP	C#
Domain 2	COBOL/Mainframe		COBOL/Mainframe	COBOL/Mainframe		
Domain 3	Mobile development		Mobile/WEB development	WEB development		

Figure 6.1. Matrix structure of the ICT organization.

By "software maintenance" we refer firstly to small changes made in a system that is running, and changes that are intended to remove errors or bugs from the software, the procedures, the hardware, the network, the data structures, and the documentation", i.e. corrective maintenance (Swanson, 1976). We also refer to major system changes like maintenance activities intended to enhance the system by adding features, capabilities, and functions, in response to new technology, upgrades, new requirements, or new problems, i.e. a modification of a software product performed after delivery to keep a software product usable in a changed or changing environment (ISO/IEC, 1999). This might be regarded as adaptive and perfective maintenance. In this study, preventive maintenance is not taken into account. There is a company-wide standard for software development. This standard prescribes the development process and deliverables, milestones, approval procedures, and quality assurance (which includes naming conventions for source code). This standard has grown out of an iterative development style. The company is also adopting agile development for some of its smaller projects, and would like to scale this up to larger projects. The standard does not dictate the use of UML or modelling in general. Enterprise-wide integration across systems is based on the service orientation (SOA) paradigm.

The tools available for modelling at the company are: Visio, Bizz Design Architect and Sparx Enterprise Architect. The tools for source code management commonly used in the software development/maintenance projects in this company are:

- JIRA: this is an issue tracking system used for bug tracking and issue tracking.
- Subversion: this is used for version management of source code.
- Tools for code coverage and code quality are also in place.

We would also like to summarise the types of documents that are commonly used in a typical maintenance project at this company. In addition, we have related each specific responsibility role to a document. This information is summarised in Figure 6.2.

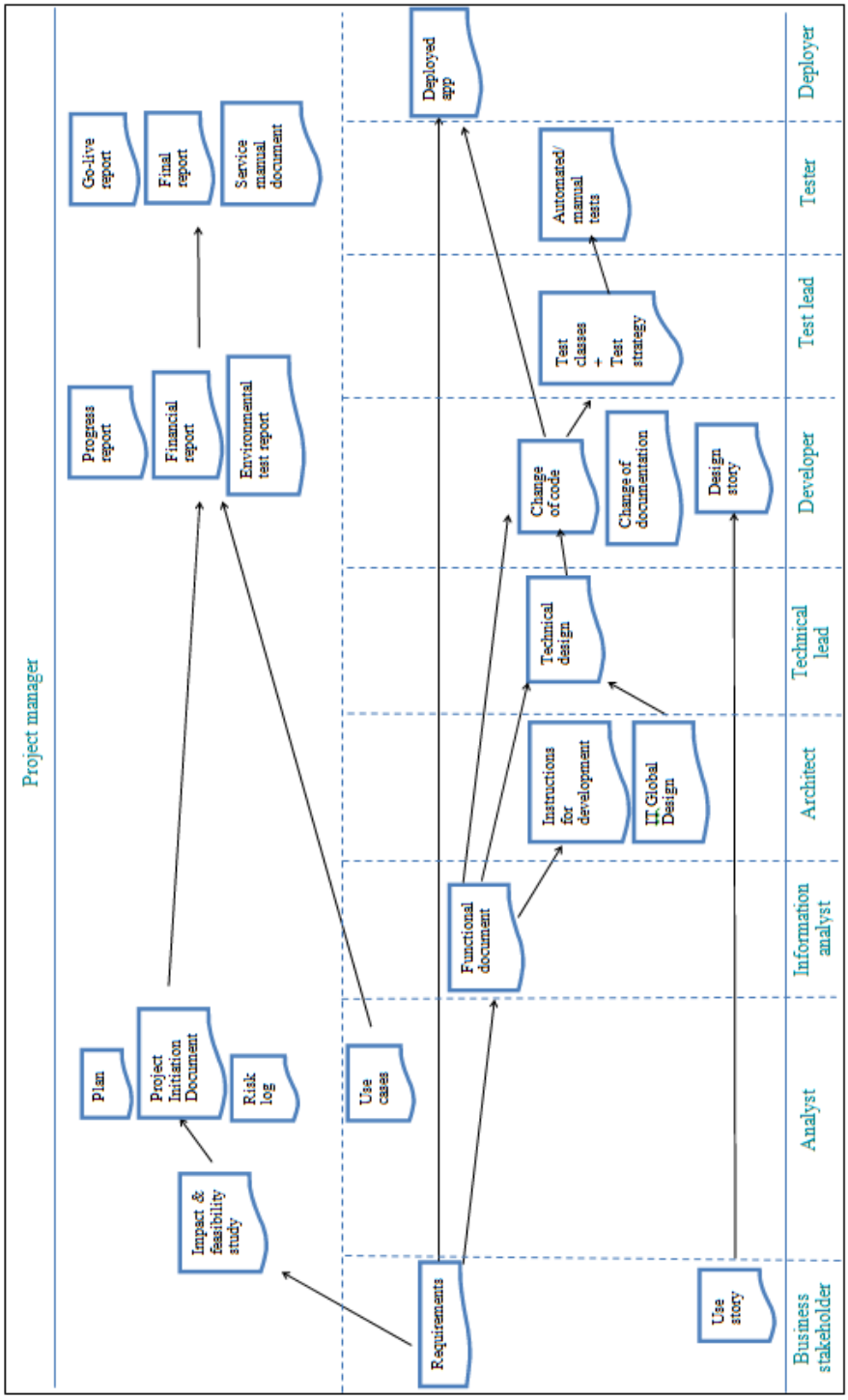

Figure 6.2. Flow of documentation in the company and responsibility roles.

Our study is a single embedded case study (Figure 6.3) and follows the classification by Yin (Yin, 2002). Our unit of analysis is the "approach to modelling". For this purpose we studied various projects within one large IT-development department (cases). In studying our cases, we quickly found out that to understand the approach to modelling, we also needed to understand the context of these projects. In particular, we had to comprehend the goals of the various stakeholders in the software development organization. Within these cases, then, we looked at modelling goals, processes, practices and tools (as the main aspects of the 'approach to modelling') from the perspectives of different stakeholders in those projects.

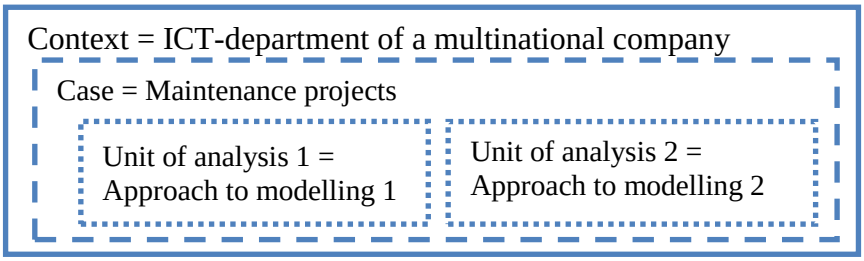

Figure 6.3. The type of case study based on Yin's definition (Yin, 2002).

We chose this company because one of the authors of this paper had arranged placements for MSc. students at that organisation, which meant that there was a collaboration relationship. The company is considered to be a fair representative of large multinational companies, having a large ICT department, a heterogeneous IT-landscape, and a long record of accomplishment in software development.

6.3.3. Data collection procedures

The source used to obtain data regarding the use of UML during maintenance tasks was interviews with Company personnel. In order to avoid trouble related to ethical issues, there was no information in the raw data that could allow a particular individual to be identified. A note was made of the names of the interviewees who participated in the case study, so as to be able to contact them when the interviews were transcribed and obtain written confirmation of their agreement to the interviews being used, but it was not possible to link their names to their responses. We used semi-structured interviews in which the interviews are "guided conversations" (McNamara, 1999). The interviews were standardised, in the sense that each interviewee was asked similar questions, but they were also open-ended, in that there was ample room for the interviewees to elaborate. We must also take into account that no choices were provided to interviewees for each question. This means that all the results of this case study are spontaneous responses on the part of the interviewees. The interview questions are shown in APPENDIX E. INTERVIEW QUESTIONNAIRE (related to Chapter 6)The 33 questions (21 main questions, some of them with subquestions) were selected using a refinement and adaptation of questions from previous empirical studies as our basis. We started with the

questionnaire of a previous survey done by the authors of the paper (Fernández-Sáez et al., 2015b). We then complemented the list of questions using as inspiration other empirical studies related to software maintenance (de Souza et al., 2005; Yamashita and Moonen, 2012). Apart from those sources, studies from different contexts, such as Embedded Software Engineering or MDE, were taken into account, but these were (partially) focused on UML (Hutchinson et al., 2014; Mellegård and Staron, 2010; Torchiano et al., 2013).

6.3.4. Case study execution and analysis procedure

The first author spent 12 months as a temporary member of the organisation in the capacity of research intern. She had direct access to the company staff, and in particular to the people involved with the maintenance projects. We first collected the data by performing 37 interviews of about one hour each. This evidence was gathered over the period 2012 - 2014. Six of the interviews were discarded because the employees were not related to software maintenance issues. In the end, 31 interviews were completely analysed, and are reported in this paper.

We started with a list of questions, because we were very interested in learning about those topics. At the same time, the interviews were done in an open form: certain questions were asked to some interviewees, and different ones were given to others. In addition, some of the questions were focussed more on one particular issue (e.g. about UML) and others were about more general topics, such as documentation in general. It should also be said that a number of insights were found that are not related to UML (the main factor under study in this case). These insights came to light during the interviews, and we allowed the interviewees to continue talking about these topics - in the spirit of grounded theory. During the research period, interviews that had already been done were also coded, in order to guide the subsequent interviews. The list of questions was used especially as support in starting a new conversation when a deadlock was reached in the existing conversation. That happened because sometimes interviewees were "shy" and did not talk too much (especially about those topics in which they did not consider themselves experts). We would like to stress that all the questions were open-questions. This meant that no choices were presented to the interviewees; the findings presented in Section 6.4 are a result of a process of:

- coding the interviewees' spontaneous responses,
- grouping the codes obtained,
- analysing the groups obtained.

The interviews were performed with people playing different roles, so as to obtain different points of view. The interviewee roles include: project manager, information analyst, project architect, technical lead, programmer or application developer, test engineer, delivery lead, SCRUM master, and systems analyst.

In order to extract findings from our study, we used the grounded theory approach, which is built upon two key concepts: constant comparison, in which data are

collected and analysed simultaneously, and theoretical sampling, in which decisions about which data should be collected next are determined by the theory that is being constructed (Glaser and Strauss, 1967). The distinctive feature of a grounded theory approach is its commitment to research and "discovery" through direct contact with the social world of interest, coupled with a rejection of a priori theorising (Locke, 2001).

The essential idea in discovering a grounded theory is to find a core category, at a high level of abstraction but grounded in the data, which accounts for what is central in the data (Punch, 2005). This is done in the following three stages (Robson, 2002):

- Finding conceptual categories in the data (coding).
- Finding relationships between these categories.
- Conceptualising and accounting for these relationships by finding a core category.

The analyst begins the conceptualisation with open coding, which is "the part of analysis that pertains specifically to the naming and categorising of phenomena through the close examination of data" (Strauss and Corbin, 1990). This is accomplished largely by asking questions about data and making comparisons, aiming to find similarities and differences between each incident, event, and other instances of phenomena (Strauss and Corbin, 1990).

Following this approach, and using the qualitative data, i.e. the data obtained from the interviews, we analysed the data in the following steps (Figure 6.4):

1. We performed the interviews, and the conversations were recorded using a voice recorder.
2. Each interview was transcribed. We used the Digital Voice Editor tool (Sony, 2010) and a text processor (Word) in order to perform this process. The transcriptions of the interviews were shown to the respective interviewees, and they either accepted them, or clarified what they had intended to say.
3. We analysed each transcription, highlighting the important and surprising statements. This was done through the simultaneous use of NVivo 10 (Richards, 1999) and Word by means of open thematic coding. This coding was carried out independently by two researchers, and then discussed.
4. We then coded the statements and grouped them under more general categories or factors (Seaman, 1999). We also used NVivo 10 and Word to perform this step.

The transcribed interviews led to a set of text files of 146,821words in total. The quantitative data was derived from the background questions that we asked the interviewees.

Finally, we summarised all the results of this case study, generating a theory. In mature sciences, building theories is the principal method of acquiring and accumulating knowledge. But there is little use and development of empirically-based theories in Software Engineering (Sjøberg et al., 2008). Theory is the means by which one may generalize analytically (Cook et al., 2001), thus enabling generalization from situations, such as case studies, in which statistical generalization is not desirable or

possible (Yin, 2002). In case-based research, we may attempt generalization from the object of study to the theoretical population immediately. For example, from the investigation of a single project, we may tentatively hypothesize a generalization about all similar software engineering projects in similar companies. Another aspect of case-based research is that variability is reduced by the breaking down of a single case into components with interactions, such as for example people and roles in a project. These components and mechanisms may be recurrent across a large set of different cases, and are hence interesting subjects of generalization. It is important to note that since a conceptual framework is a set of definitions, it cannot be true or false, but it can be applicable or not (Wieringa and Daneva, 2015).

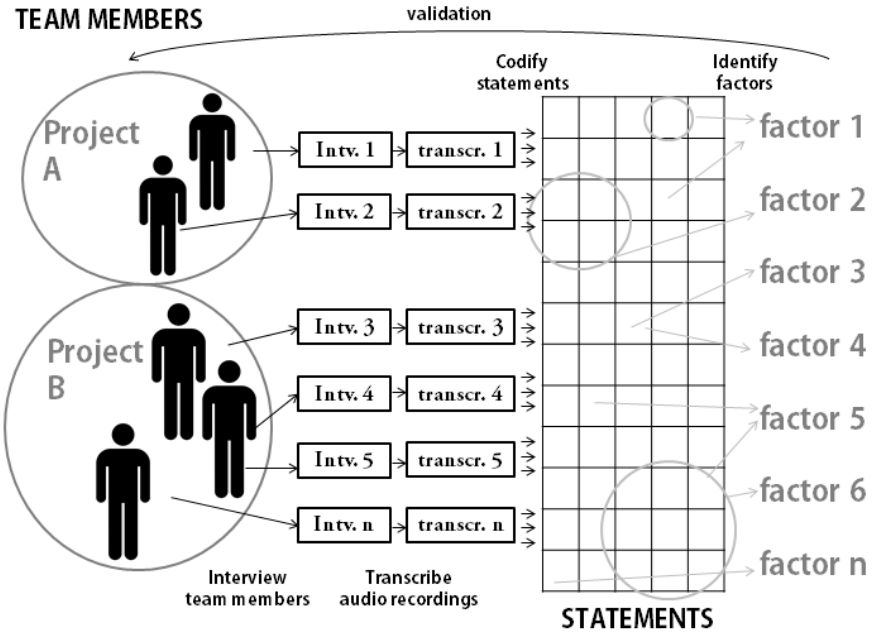

Figure 6.4. Case study execution and analysis procedure.

We decided to generate a theory as the final result of the grounded-theory process, motivated especially by the comment of Johnson et al.: "many proposed [...] methods, programming languages and requirements specification languages exist, but very few explicit theories explain why or predict that one method or language would be preferable to another under given conditions" (Johnson et al., 2012). It is not worthwhile to develop a general theory of software engineering, but it is very useful to develop incompletely specified, partial theories that can be applied to practice (Wieringa and Daneva, 2015).

6.4. RESULTS

This section presents the results of our case study. The main results of this empirical study are qualitative data, although in some cases we have provided a few

statistics when these help to reinforce the importance (by repetition) of a specific finding. In order to present all of them in the best way, organizing them and linking them when possible, we have produced a theory inspired by the elements highlighted in (Sjøberg et al., 2008), along with the authors'(previous and current) observations of industry practice. Our theory is formulated on the basis of conclusions drawn and extracted from observations in industry, the objective being to capture all the observations of this particular case study and help explain them.

The diagrams of our theory contain the *constructs* (*what* the basic elements are), and *propositions* (*how* the constructs interact). The *explanations* (*why* the propositions are as specified) of the theory are detailed in the text accompanying the diagrams. Note that the *scope* (what the universe of discourse is in which the theory is applicable) of the theory is the same as the context of the case study. We used the nomenclature suggested in (Sjøberg et al., 2008).

The theory is presented by means of multiple diagrams that focus on different areas of the theory. Figure 6.5 presents the "baseline" theory. On it we represent the environment in which the results of the case study will be embedded. On the left side, we can observe elements related to a typical SE approach. We selected only the main elements that we found in our case study, but there are definitely other elements which might be good candidates for being represented also there. On the right side we represent the approaches of interest in this case study: the approach to modelling (and especially the approach to UML modelling), and also the approach to documentation (for example when modelling is not available).

The main statements represented in this theory are the following:

- Projects take place in a context, have stakeholders and can be in a particular stage of a lifecycle.
- Stakeholders have goals, and these may change throughout the execution stages of a project.
- The stakeholders' goals drive the SE process used and the practices used in the overall approach to SE.
- A process denotes the collection of (formalised) steps of tasks that the project follows to engineer software.
- The processes and practices in turn drive the choice and use of tools.
- Part of the overall approach to SE is the approach to documentation (AtD) and the approach to implementation (AtI). AtD and AtI refer to a combination of goals, processes, practices and tools for documentation and implementation, respectively.
- Together, the AtD and AdI drive the approach to modelling (AtM). The approach to modelling itself again consists of a goal-modelling process, a set of modelling practices and a collection of modelling tools. The modelling approach (which is based on UML or another notation) represents a "bridge" between the AtD and the AtI.

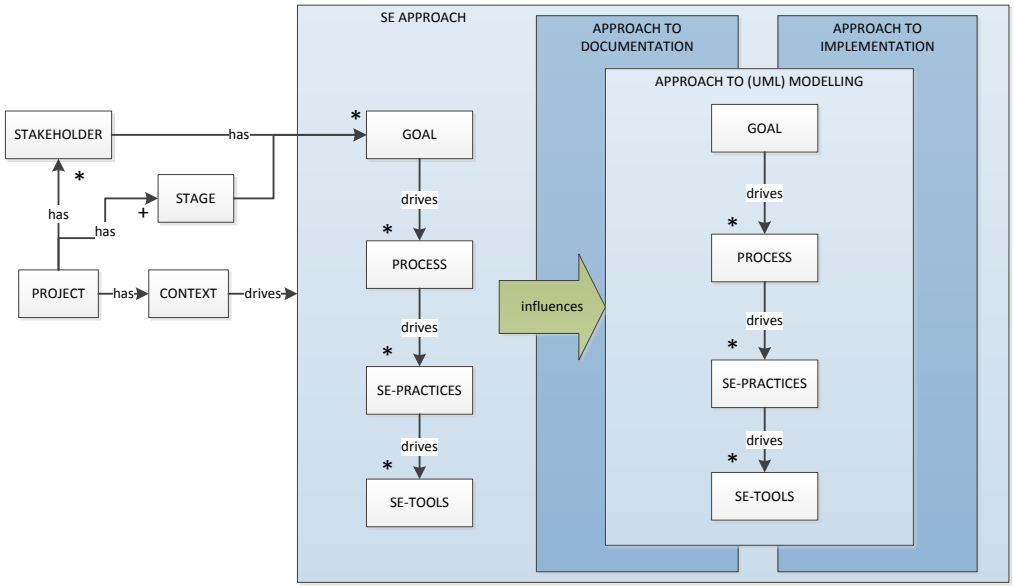

Figure 6.5. "Baseline" theory.

We would like to point out that the boxes containing others do not necessarily mean an inheritance relationship; a "part of" relationship is more appropriate in this case. The legend for all the diagrams that present our theory is summarised in Figure 6.6.

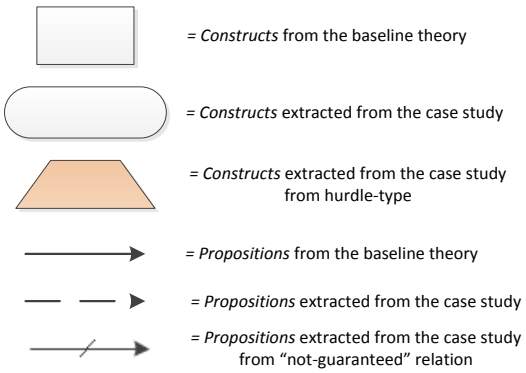

Figure 6.6. Legend of the theory-diagrams.

We present the results of the case study, grouped by each element of the SE approach represented in Figure 6.5, and we also made an effort to relate them to each RQ. As mentioned above, all the results in the following subsections were extracted using the grounded theory methodology.

Furthermore, the results of this case study are complemented with quotes from the interviews that were considered noteworthy as regards the topic being analysed. Those quotes are labelled using the term "Int", followed by the number of the interviewee (for example, [Int12] would be used to label a quote from interviewee number 12). A summary of the main background information related to each interviewee is presented in APPENDIX F. We ask the reader to take into account that a few of the interviewees did not want to provide some of their personal details (educational level or field of education). Although it was emphasised to them that the interviewees were going to have their identity kept secret, these individuals thought that their responses might make it possible to link to them to those more personal data, so they preferred not to provide the personal details mentioned. It is also important to point out that since not all the interviewees responded to all the questions, the percentages shown in this paper were calculated in two ways:

- Relative to the total number of interviewees.
- Relative to the number of interviewees that mention this term/category.

The means used to calculate the numbers is stipulated in each section.

6.4.1. Background

We asked the interviewees to fill in a short questionnaire about their background in order to characterise them, but not all of them wished to give this information.

With regard to the interviewees' gender, 10% of them are female and 90% are male. This proportion was expected, based on our personal perception of the proportions of ICT employees at the company.

The interviewees' educational level would appear to be mostly (11) Bachelor Students, e.g. polytechnic level, and Master's Students, at universities (4). The fields of education they come from are mainly from Computer Science degrees, although there are other fields, such as Arts, Business and Finance, Chemistry and Physics, Electronics, Maths, Psychology and the Navy.

We also asked our interviewees about their experience in ICT. We classified them into the following categories:

- **Low experience**: those who have been working in ICT for less than one year (2 interviewees).
- **Medium experience**: those who have been working in ICT for between one and five years (2 interviewees).
- **High experience**: those who have been working in ICT for between five years to 10 (4 interviewees).
- **Very high experience**: those who have been working in ICT for more than 10 years (19 interviewees).

The majority of the interviewees are very highly-experienced people.

The roles of the interviewees were varied: analyst developer (1), delivery lead (1) deployer (1), information analyst (3), program analyst (1), programmer / application

developer (9), project architect (5), project manager (2), SCRUM master (1), system analyst (2), team leader (1), technical lead (2), test coordinator (1), and test engineer (1).

Half of our interviewees (16) responded to the question about the projects on which they were working at the time of the interviews (the others preferred not to say, as they were concerned about privacy). The types of projects were varied:

- Common projects, concerning desktop or web applications (31%)
- Mobile application projects (25%)
- Projects regarding the maintenance of old legacy systems (13%)
- Outsourced/offshored projects (13%)
- Projects concerning mainframe systems (6%)
- Projects regarding migration (6%)
- Projects related to embedded real-time programming (6%)

6.4.2. Goal

In this subsection, we present the results of our case study that are related to the element "GOAL". Firstly, we focus on the different goals or purposes of using UML, all of them mentioned by the interviewees. This subsection contributes to answering RQ1. We also present a subsection about "costs," which is a specific goal (to reduce the project costs), and common to the majority of maintenance projects. That subsection contributes to answering RQ2.

6.4.2.1. Purpose of use of UML

One of the questions during the interview was: "*Why do you use UML diagrams? / For what purpose is UML modelling used?*" The answers to these questions varied, and are shown in Figure 6.7. When reading the figure, please bear in mind that one interviewee might have mentioned more than one purpose, so it is important to note that the percentages represent the ratio of interviewees who spontaneously mentioned a purpose (they do not add up to 100).

The majority of the interviewees use UML as a communication tool (63%), in line with other empirical studies (Petre, 2013). This communication can be:

- Between team members (38%).
- With members of other teams (9%). This is especially important when part of a team is geo-distributed, or when part of the maintenance is outsourced. In these cases, the use of prototypes would also be very helpful as regards properly communicating what should be done.
- With stakeholders (6%). This type of communication is usually difficult, and UML helps to establish a standardised means of explaining the size and the complexity of the project to stakeholders.
- UML is also used to communicate the current situation to newcomers to the project (16%). This is very important in the quest to avoid "knowledge evaporation". That

situation occurs when an expert on a system leaves a project, taking with him/her some knowledge that is not documented.

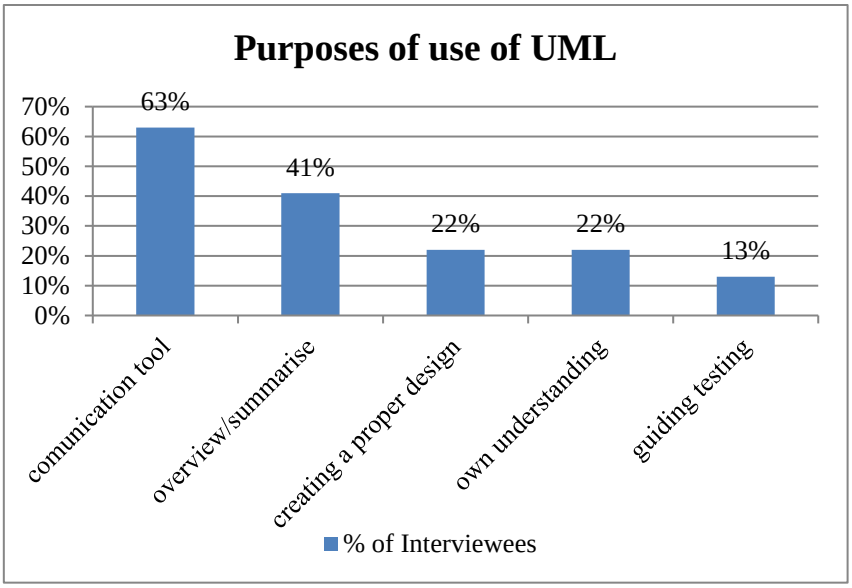

Figure 6.7. Responses regarding the main reasons for the use of UML.

The wide use of UML as a representation for communication might be thanks to the fact that it is a standard notation, and because it is well known both by professionals and recent graduates. People also recognise that UML diagrams are used to complement verbal communication (face-to-face or written), but not to replace it: "*UML helps to improve the communication, but it doesn´t replace it*" [Int4].

The next most common use of UML diagrams is to obtain an overview of the system being maintained, or to summarise it (41%): "*UML is mostly a high level picture or presentation of what the landscape looks like; so, by that I mean, this is a product which is placed in this domain; it uses these services, these data models, and they interact with each other in this way*" [Int6].

The next most common use of UML is to help to create a proper design (22%): "*UML is a sort of a tool you can use while thinking about and creating things; going over them again, you'll find mistakes in the next step; you can adjust new elements, new classes and…so it's also a sort of brainstorming that in that way supports, really supports, the specification process*" [Int35]. With regard to sequence diagrams, they are used to help plan for a solution, but the diagrams are not created for documentation purposes: "*We have often seen that developers start coding without really thinking through the programme that they have. [..] However, having a sequence diagram in place will help them to think through the problem before they solve it. [..] Sometimes, depending on the need, it may be kept or discarded, because when you cannot maintain it you discard it, but the purpose is actually fulfilled*" [Int24].

Another purpose of using UML which is often reported as much as the previous one (22%), is to enhance people's own understanding of the system being maintained. A modelling notation is also a tool which can help maintainers to create their own mental representations of the system being maintained, as well as of the task that needs to be carried out: "*When you pick up something new, big, and challenging, then you start writing your thoughts on the whiteboard, and at some point on the whiteboard, the clouds converge to show something that is structured. So you make a diagram of it. You think it out. Then you can you back to the whiteboard and do the next level of understanding. And that is what is great about UML. […] You can use the diagram easily to explain what you have in mind. That is what I usually do*" [Int4].

When comparing these findings to related work, a question arose: how closely do these internal structures correspond to external representations, whether these are in relation to program code or to visualisation tools? This question was studied by Petre's work (Petre and Blackwell, 1999), but her study is more oriented to the designing of program codes, rather than to architectural design. In our interviews we find that UML is used both by architects and programmers. Petre reports that software engineers use 2D, multidimensional and multi-model mental representations.

The next most cited reason for using UML was to guide testing and to plan the rest of the project, creating a to-do list (13%). As an illustration, we quote: "*Sometimes I put in things just to remember that these are aspects that are also there, or […] I put in the components that do not exist or seldom exist, just to remember <I have to do something with this>[…], then at least we don't forget to address that aspect at the very beginning*" [Int3]. Some project managers also use UML diagrams to keep track of progress [Int37].

Uses that were mentioned, but only rarely (3%; i.e. by only one person for each use), include: guiding implementation, analysing risks, documenting, following the mandatory process, justifying costs, supporting maintenance, determining responsibilities for success (offshore team), monitoring implementation, having a professional way of developing, checking the quality of the implementation, or showing progress. This list of rarely mentioned purposes of the use of modelling is aligned with the results of the survey by Liebel et al. (Liebel et al., 2016). The study in question focused on embedded software systems and discovered that the main purposes of using models in that domain are: simulation, code generation, and documentation. This makes sense because the automation of activities in the embedded domain during the development process would appear to be of great importance.

We should also comment that some possible purposes which we expected were not mentioned by any of the interviewees; these included purposes such as certification, deployment, generation of implementation, knowledge transfer or reasoning about design.

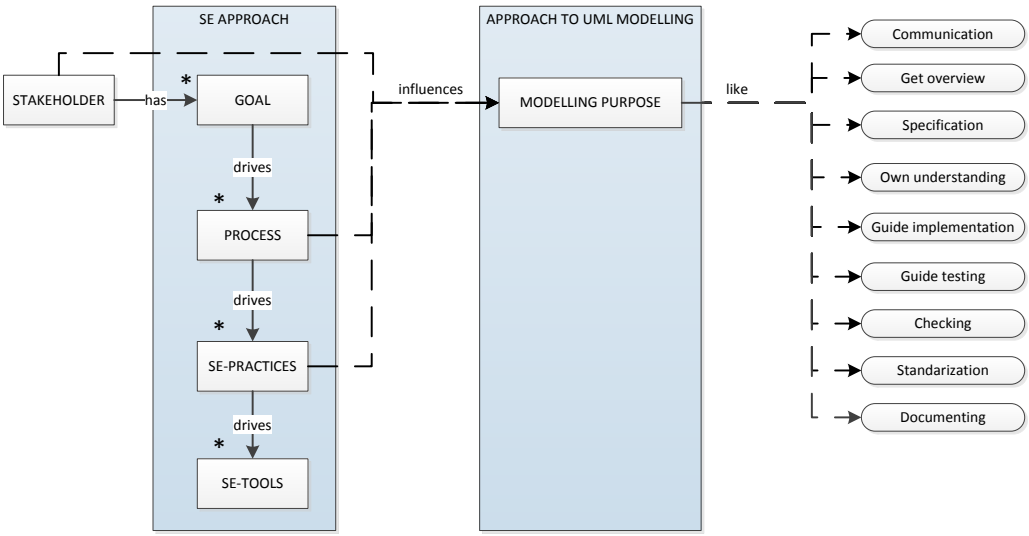

Figure 6.8. Theory summarizing the purposes of the use of UML.

The responses to this question show that – when available - UML models serve a variety of purposes. The main purposes are rather 'soft/fuzzy' and are related to communicating, understanding and creating a design. Figure 6.8 shows the main purposes of the use of UML, along with the elements that influence it. In this figure, and in the other figures concerning the theory, the dashed lines represent evidence-based relationships. These relationships are represented using arrows, which link an element A with an element B. For example, the "influences" arrows mean that a decision in A has an impact on decisions in B. The "like" arrows mean that B is an example of A. The remaining names given to the arrows could be considered as self-explanatory (for example, "increases", "leads to", etc.).

6.4.2.2. Perceived cost factors of modelling

We asked the interviewees about the possible cost factors or investments related to the use of a modelling notation like UML in software maintenance: "*What cost factors are related to the use of UML modelling in your work?*". Table 6.2 shows the responses to this question, and their ratios; these were spontaneous responses, i.e. the costs mentioned in this section were not suggested to the interviewees at any moment. The majority of the interviewees considered that training is an important investment.

The types of training costs mentioned could be split into two main groups. One significant factor that emerged was the cost of training in the UML notation. So we can say that in some sense the educational background may influence the discussion on the training cost of using models. This might be due to a fear of their own poor understanding of UML: "*They assume they understand the entire diagram, but the fact is that they completely misunderstand the diagram. So it depends on how you do the communication*" [Int3]. During the interviews we detected that the term UML is

sometimes considered to be a synonym of Rational Unified Process (Jacobson et al., 1999) or even Object Orientation (OO) (Blaha and Rumbaugh, 2004; Bruegge and Dutoit, 2010). Some of the interviewees thought that the use of a standard notation is not sufficient, and that training is necessary to establish alignments and conventions. It is also noteworthy that several engineers (especially those that have not passed through education in Computer Science or Software Engineering from a university), stated that they learn by doing, and that they train on the job. However, if they learn by looking at existing documentation, they will learn 'box-and-line' diagramming, with very little knowledge of the variety of syntactical elements, or of their official meaning.

Table 6.2. Cost factors related to the use of UML.

Cost factor	% of interviewees
Training	48%
in UML notation	32%
in modelling tool	16%
Tooling	32%
Migration of documentation	10%
Change of people's mind	10%
Change of process	6%
Cost of not updating	6%
Cost of creation and updating	6%
Central governance	3%
Learning curve	3%

To discover the actual costs related to training, we obtained historical data, provided by the person who manages internal/external training and courses for employees in the ICT department. We used historical data from January 2006 to May 2012. We were able to select those courses which were related to training in the use of UML, and separate them from other related topics (like Object Orientation, RUP, etc.). The total amount of money spent by the company on UML adds up to 24,313€ in a period of six and a half years (which is approximately 3,750€ per year). A fair number of interviewees mentioned that they did not need a course in UML, because they had had training in the use of UML during their tertiary education. This amount of money invested on UML training is small, compared to the total budget of the department.

On the other hand, a good number of interviewees did consider that training in the use of a modelling tool should be a potential cost factor. They also thought that the lack of training in the use of tools would mean a loss of the possible paybacks that the use of the tool might bring. In the ICT department, 80 people had been trained in the use of a particular UML tool. Buying/licensing a tool is reported as a cost factor by

32% of the interviewees. Based on data from the company, we can calculate that tool-licensing costs around 18,000 Euros/year.

Another investment which was often mentioned by interviewees is the cost of the 'migration' of the current situation (especially of documentation) to the new one; i.e. updating the existing documentation to include UML diagrams where there are none, or where another notation is used. Formally speaking, this is related more to the introduction of UML than to the use of UML, yet it is potentially a major investment for companies that are maintaining a large set of systems. Most comments related to migration came from people who are currently working on non-UML projects, and who would like to introduce UML, but who consider the migration of the documentation to be an impassable hurdle. Figure 6.9 summarises these findings and relates them to our theory.

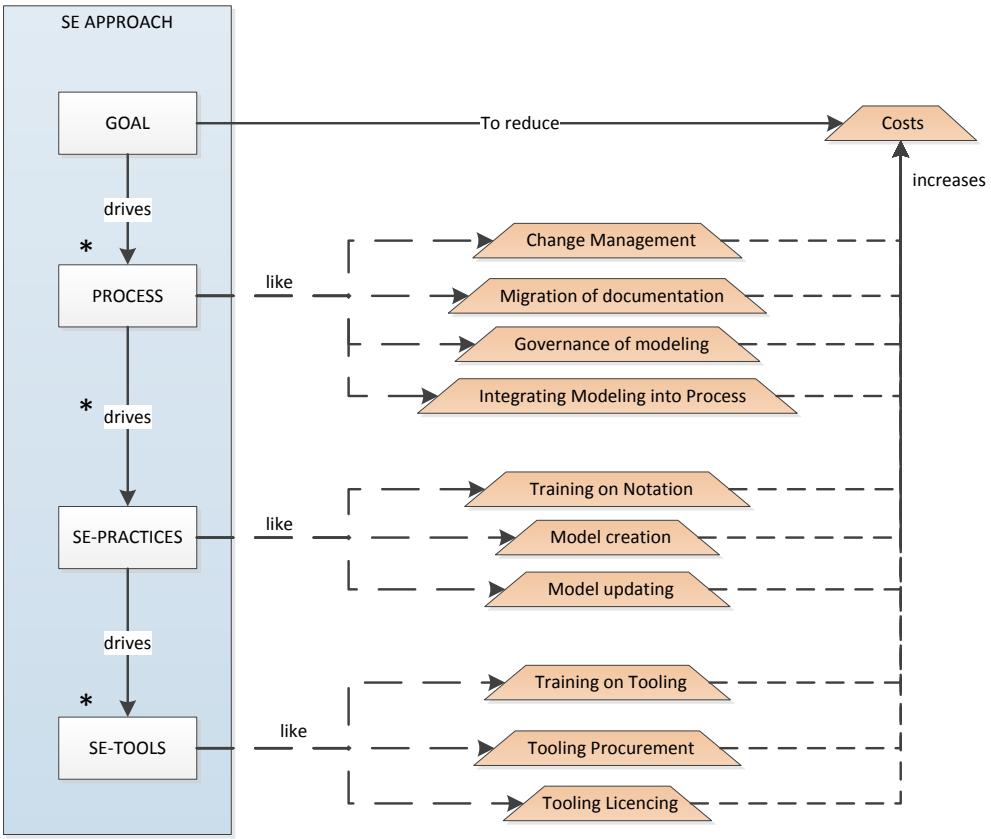

Figure 6.9. Summary of findings about cost-types.

In addition, the interviewees mentioned other costs related to the presence/absence of UML diagrams; these costs are less tangible. Furthermore, the interviewees stated that projects incur costs also because of poor documentation and modelling.

- Miscommunication: although there is a common knowledge of a standard modelling notation (based on training at Universities, or specific training at the company), there are also costs related to miscommunication: "*During my study I had a course on UML, but I think it was a bit outdated, although the basics are still the same. Then I started work in this department which implements UML too, in part, so you do get to know different diagrams and how to use them. But even though I think most people have received the same training, there is still a lot of miscommunication, often. So even having a formalised diagram does not necessarily mean that everybody is on the same page. If you look at them all.*" [Int14].
- Lack of modelling: When there is no up-front design available from the beginning of a project, there are greater probabilities that a refactoring later will be needed later in the project, because the requirements were not completely understood. The cost/penalty for repairing/refactoring the implementation is more expensive than doing lean up-front design: "*I'm convinced that if you did not use UML in the course of a big project, you would have to do a lot more refactoring; so in UML diagrams or in graphical diagrams, the structure of what you are going to do becomes quite clear and you can easily see if there are things missing. It's a little bit like you have a blueprint and you are going to build a building; you need a blueprint, to tell you what to put into the foundations, and then what to establish what you have to do after that, and after that, and after that. It determines more or less the order of your development too. If you don't have a plan of your building, and someone starts to build, it will at some point collapse for sure.*" [Int26].
- Moreover, when no up-front design is available, the probabilities of defects finding their way in are higher. If a defect that leads to operational defects is left in the software, then this triggers a chain of activities that also involves time from the helpdesk. That may also have an impact on business processes: *"… there is no upfront design [..] we directly jump to the code, we test everything. That's how we started. We failed miserably.*" [Int24].
- Lack of up-to-date documentation: One of the comments frequently made in the interviews was that that people (especially developers and testers) needed to call other people in the organisation to discover whether the documentation was up-to-date, because they were not sure if that was the case [mentioned by Int36]. This leads to costs in effort, but also to significant delays in time if the person cannot be reached immediately. This lack of one of the most critical quality attributes (up-to-date-ness) is aligned with the results of previous studies (Garousi et al., 2013). In relation to this, we asked some of the individuals interviewed about their preferences for software documentation. We posed them a hypothetical dilemma: They would have to choose between two different projects: the first one is a project which contains UML diagrams in the documentation, but the documentation might not be updated; or the second one, which is a project with updated documentation but without UML diagrams. Two thirds of them chose the UML project, while the rest (one third) selected the updated project. Those who chose the updated project

argued that UML diagrams could be generated easily. Those who preferred the UML projects argued that they prefer to have a quick overview of the system and then go to the code for details about the current situation.

A summary of the perceived cost factors of modelling is summarised in Table 6.3.

Table 6.3. Summary of cost of modelling or not modelling.

Costs of Modelling	Magnitude of cost	Costs of not modelling	Magnitude of cost
Training	Low	Possible misunderstanding	Medium
Tooling	Low	Spending time reading code	Low
Creation and update	Medium	Possibility of defects being incorporated	High
Migration	High	Need for refactoring	High
Change people's mind	Low	Cost of reverse engineering	High/medium
Change process	Low		

6.4.3. Process

Here we present the findings related to the element "PROCESS". We provide the interviewees' opinions on the relationship between their development processes or SE methodologies and the presence of (UML) modelling. This subsection contributes to answering RQ1. After doing that, we delve into how documentation is used as part of the process of maintenance of a system. Firstly, some general hurdles in relation to documentation are highlighted (this contributes to answering RQ3). We then summarise how the documentation of a maintenance project is used, looking at its usability/usefulness and how it is maintained. It should be borne in mind that we are talking about documentation in general in these subsections, but UML can be considered part of that documentation, so the findings can be extended to apply to UML also, and in fact some of them are specific to it.

6.4.3.1. Relation between development process and modelling

During the interviews, we found that the approaches that are principally used at the company are the waterfall and agile approaches, like SCRUM (75% of the projects are waterfall, 25% agile). In this subsection, we share insights into the maintainers' perceptions when we asked them about the relation between development processes and modelling:

In waterfall projects, there is a lack of communication or discussion. The information flows in only one direction, and some information is lost on each step. Waterfall approaches therefore lead to less effective solutions: "*If you do pure waterfall style, somebody gets a design approved and then gives it to some other guy, and the other guy says: 'Well I looked at your design, but it's not going to work.' And then the first guy says 'Hey, it was already done and it has been approved, so you have to build it this way.' Yes, but it will not work; so there's no discussion there,*

which is an even more ineffective approach than if you were to talk about it first." [Int16]. The perceived quality of agile projects therefore seems to improve in comparison to waterfall projects, on the basis that: 1) everybody agrees on the solution and 2) there are multidisciplinary points of view when constructing the solution. It is thus possible to state that people are more open to revising the design in agile projects.

One interviewee mentioned that the use of UML is not compatible with agile projects, and he would not recommend using it in that kind of projects [Int6]. Nevertheless, his colleagues did not agree with him, because they considered that modelling is also a good practice in agile development. Modelling can also be carried out in an incremental manner [Int24], following the philosophy of agile projects, meaning that using UML does not imply a big design upfront.

We also detected that there is sometimes an incorrect usage of the development methodologies when there is an effort to save time. This leads to a mix of waterfall and agile projects: "*In waterfall we have the big design upfront, so first you do a full initial analysis, and then design; when everything is clear you start to develop. That's the most formal approach in waterfall, but nobody does that. So when a project starts, the waterfall does document, but the early stage of development then starts on the basis of the things that are clear at that moment, so we try to mix a little bit of waterfall and agile. But it's not the ideal situation*" [Int26].

It was also detected that agile projects suffer from a lack of documentation. The main documentation is carried out directly in the source code as comments. This is aligned with previous research (Garousi et al., 2015), in which maintainers have stated that they prefer to refer directly to the source code itself and rely on source-code to support their information needs for maintenance tasks. This way of working implies difficulties for newcomers to the project. There is also a clear need to name conventions and use self-descriptive names in the source code. These problems are magnified when the code gets bigger and more complex. The interviewees considered that the lack of documentation is complemented by a face-to-face communication and by obtaining more experience on the same project [Int33]. But in the agile projects, maintainers also argued that the updating of the documentation should be part of your 'definition of done' [Int24], otherwise it would never be done.

These findings about the influence of the SE process in the modelling approach are summarised in Figure 6.10.

Finally, on this point, we discuss some remarks regarding the relationship between modelling and the different phases of a common software life cycle. Modelling is usually done during the early stages of software development. There is a clear relationship between the production of source code (during development or during maintenance) using all the available documentation, including the presence of UML diagrams, because UML diagrams are used to guide the development of the software. The interviewees also stated that there is a clear relation between modelling and the software testing phase, because UML diagrams might be used to guide the testing use cases. They considered that one of the main issues of testing is to check the alignment

of the source code with the diagrams. If the source code reflects what was represented in the diagrams, the quality of the system is ensured.

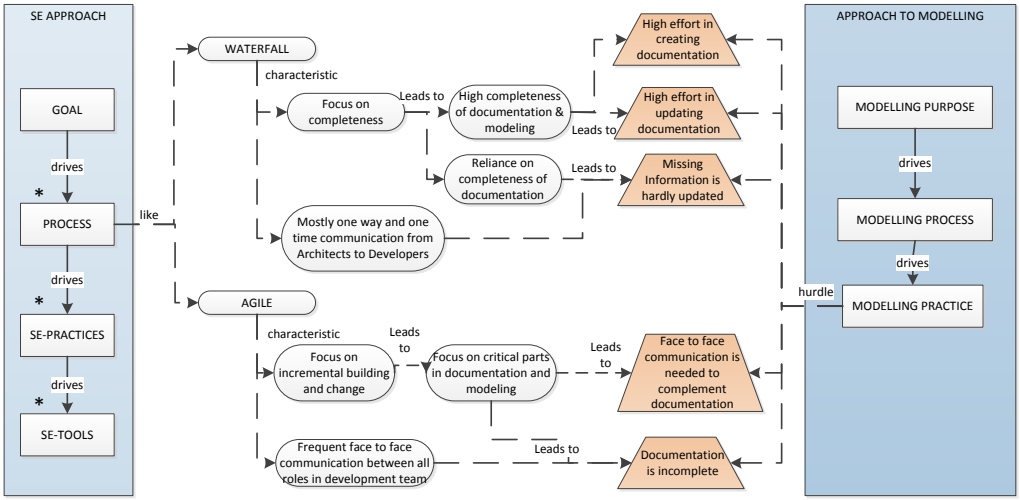

Figure 6.10. Influences of the SE process in the modelling approach.

We also found problems in the investment made during the different parts of the software life cycle. Very often, the investment made in the development phase is insufficient. The reasons for this could be:

- There is a lack of time to finish a project punctually, which means that the construction starts before the end of the definition of the requirements or before the completion of the modelling of the system.
- Sometimes companies that work as suppliers make a bid on a project and allow low development costs in order to 'win' a project when competing with other companies. After doing so, this supplier company builds up unique expertise in a particular system and is then in a good position to also be awarded many subsequent maintenance projects. The supplier company therefore acquires a stream of maintenance projects.

These two practices are carried out following "unwritten" rules of the company, although the interviewees mentioned that solving a problem in the design phase is cheaper than solving it later, during the maintenance.

Furthermore, there are projects which invest all their resources in the construction of the system while the company simultaneously attempts to invest a minimum amount in the maintenance of these systems. As a result, such systems end up being extended in a quick-and-dirty manner, by means of "patches".

All of the above leads to longer maintenance projects that are sometimes difficult to change owing to the ad-hoc means used to construct the system. We would therefore like to state that cheap developments sometimes lead to unmaintainable systems.

6.4.3.2. Documentation

Although our interviews focused mainly on UML, some interesting conversations about software documentation in general also arose. This was due to two factors: 1) we treated UML diagrams as part of the documentation of the system, and 2) those who did not use UML talked to us about documentation in general. The responses to those interviews allowed us to extract several hurdles that hinder the effective use of documentation in general. The majority of these hurdles (80%) show that the use of modelling is not only a problem of modelling itself (per se), but also of the overarching technologies used to handle and navigate documentation:

- There is a high level of duplication of documentation in different storage systems: Alfresco, Jira, Confluence, Dropbox, Shared folders, network driver, etc. This makes the documentation difficult to manage, to search in, and to update. A reduction of storage facilities and standardisation would be needed. Otherwise finding documentation for a system is frequently a time-consuming activity.
- Different projects use different structures (directories, naming) for stored models (and often files in general). This again complicates the accessibility of information. Standardisation would be needed to solve this problem.
- People experience difficulties in searching for a specific document/information. This might be caused by the two previous points on this list. The problem related to the fact that the information required is believed to be contained in available documentation, but cannot be found has already been highlighted in previous research (Lutters and Seaman, 2007). Information that is needed is often 'buried' in, and frequently scattered throughout, voluminous documentation.
- For ongoing projects, their documentation may not yet be available in the shared repositories, while peer projects might already need to consult it.
- Techniques used for cross-referencing between documents are not reliable (there are sometimes shortcuts to non-existent paths, so they are 'dead links').
- Once the documentation is found, the reader needs to verify that the documentation is up-to-date, and consumers therefore need to spend time (especially making telephone calls) verifying whether documentation is up-to-date.

Each of these issues by themselves constitutes a critical hurdle as regards ensuring that documentation is kept up to date - critical in the sense that if the documentation cannot easily be found, the engineer will not bother to use it and will therefore not update it: "*So of course it's not always difficult to get the data, but then you really need to know the people. That means that you are depending on people to get information, instead of depending on a good system, a good document system where you get the data yourself. That's the thing.*" [Int32].

Storing all the documentation in a single document would not be usable, because dividing documentation is necessary for reasons of size and abstraction. However, this division is hindering the practical use. The partitioning of information across multiple documents makes it more difficult to keep information consistent and up to date: "*I hate jumping between documents. That makes it hard to have an overview of what's*

what" [Int12]. Another reason for dividing the documentation is that of presenting different views to different stakeholders: "*The core development team consists of 3 developers, one information analyst and one project leader; hence 4 'technical' people. In addition to that, there are eight different stakeholders involved in the project (deployers, databased admin, maintenance, business analyst, or project architect). Some of them are informed; others need to give different types of approval. Now for whom should the documentation be optimized?*" [Int13]. This practice was also detected in previous empirical studies (Petre, 2013).

Even if a document can be found easily, there is another problem: there is strong agreement amongst the interviewees that it is hard to find the relevant information within a document:

- Sometimes it is difficult to find information within documents. This problem should be solved if we are to facilitate the effective use of documentation.
- For example, business rules are not put in UML, but they do influence the design [Int26].
- Often the 'why' (rationale) is missing [Int23, Int27].
- It is not easy to search in diagrams/models [Int37, 28].

Another problem detected, also related to the documentation is the fact that it is based on projects rather than around systems: "*You would like to have a distinction between system documentation and project documentation, and both have a valid position throughout the documentation, but that is not really the case at the moment, unfortunately*" [Int1]. This increases the traceability of changes per release, but it complicates the understanding of the system as a whole. "*Well, it depends; for example, project documentation [..] nobody really reads it, as there are only a few documents that are really interesting, so…on one hand there's too much documentation [..]. So for the….to keep documentation, [..] if a process is changed in the business or if a software is changed, not always has the documentation also been changed, so I think there should be a way where you say <<Ok, this is part of it as well, so change the documentation too, including a review, so that it's better managed>> not only manage the system but also the documentation. There I see a lack sometimes, but the project documentation is sometimes just too much. ….. That usually takes too long; it's too much, and nobody reads it*." [Int32]. What is more, processes do not enforce the updating of documentation when software is changed. This is supposedly compulsory for all changes, but it is completely unmanaged as things stand. Having some type of process control could be good (and would be easy to implement in a tool).

As a reflection which could be extracted from the problems mentioned in this section at this point, we would like to point out that the use of modelling is not only a problem of modelling itself (per se), but also of the overarching technologies used to handle and navigate documentation.

In this section, we have already discussed several issues related to the documentation of a project. Nevertheless, there are other issues containing sufficient

entities to be treated independently. In the following subsections we would therefore like to highlight findings related to the maintenance of documentation; these are findings about how the documentation is being used as part of a project, as well as about the usability/usefulness of the documentation itself.

6.4.3.2.1. Maintenance of documentation

We also asked the interviewees about the maintenance of the documentation (in general, not only the diagrams). It is very surprising that the maintenance of documentation took place in only 39% of the projects on which the interviewees had been involved, although this result is aligned with that obtained in a previous survey (Forward and Lethbridge, 2002). The interviewees stated that some of the reasons for this might be the following:

- It is not clear who is responsible for maintaining the documentation.
- There is a lack of compliance to processes. In addition, there are sometimes organisational or process problems blocking the maintenance of the documentation.
- There is a feeling akin to "don't care' about maintenance". This is an attitude issue, and maintainers considered the documentation to be a low priority task. When updating documentation is not a task that is enforced, it will be relegated. This has to do with the fact that the benefit of documentation is felt by people other than those who create the documentation; the creators of the documentation therefore consider it a burden.
- Small changes are not represented in the documentation (especially in the diagrams). Small changes could be maintained in the text of a document, but it is very unusual to have the need to update a diagram. Where a diagram might be changed for a small maintenance task, it would be the sequence diagram. The same occurs when a technical modification is made. That means that only structural changes (which are not frequent) are reflected in the documentation.
- There is a lack of time, which does not allow team members to "spend time on secondary tasks" (time pressure). Sometimes they need to make "urgent" changes, and these are not reflected in the documentation. If no time can be found to update the documentation while the maintenance of a system is being carried out, it would be useful to update the documentation after the project has finished. However, there are no incentives to report out-of-date documentation, or any processes with which to do so.
- Team members tend to memorise the systems and leave the documentation. They do not need to update the documentation, because all the information is "in their heads". This then leads to a problem of knowledge evaporation when a team member leaves the project: "*I think lots of information is also in the heads of the people working there, which was also the case in our mainframe department, but there was always a focus that it should not be only in the heads of people, because if people go away ...*" (a perspective from an information analyst and developer who moved from waterfall to agile) [Int35].

- Some interviewees argued that the stakeholders do not want updated documentation. They only pay for a maintained system, and they do not care about the documentation. In such a setting, a team finds itself applying all its resources to the source code, rather than investing a small part in maintaining the documentation.

Several reasons were given for not maintaining the documentation, but the maintainers also know that not doing so has several risks:

- Documentation which is not updated becomes an incorrect basis for the maintenance in the future "[..] but in the end it becomes a problem because no one knows how the system works, and so some developer needs to look into the doc and from the doc, to determine how it works. So then at some point, the diagrams and the code work totally differently; then you have a problem with everything, with your maintenance, because existing diagrams are also determined to be a basis for making a code for further enhancements of a system" [Int26].
- Non-updated documentation might lead to mistakes becoming incorporated into the source code "For example, let's say one year the team does diagrams and the code religiously; they give them 'in sync'. The next year, for some reason, nothing is in sync, due to project pressure, and the 3rd year, if someone wants to refer to the diagram and then goes on with coding, there will be issues. So yes, it is helpful if kept in sync with the code" [Int37].

Another important point is that when the documentation is not properly updated, problems related to low traceability between software artefacts could arise "*I'm reading this but it does not correspond to what I'm seeing on the screen. And the only thing we have left is to ask a developer how this piece of functionality is implemented, because I cannot find it anywhere. After this we can make a new specification, but it really also delays both the maintenance track and the project*" [Int14]. This finding agrees with the survey of (Nugroho and Chaudron, 2008), in which respondents highlighted the importance of maintaining the correspondence model-code. They argued that, from a software maintenance perspective, we lose the main benefits of models as an important source of architectural information when software models no longer correspond to the implementation code.

The reasons mentioned for not updating the documentation (including or not UML diagrams), and the risks of skipping this step are summarised in Figure 6.11.

We conjecture that the difficulty involved in updating the documentation has an inversely proportional relation to the level of detail (LoD) of the documentation. When documentation (or a diagram) is represented using a high level of detail, the understanding of the system is increased, but the ease of updating it is decreased, and vice versa. Projects need to find their 'ideal' level of detail which might balance those concepts. Different levels of abstraction in documentation are found because of the fact that different versions of the documentation are created for different purposes. Low level of detail is created to satisfy the mandatory process and also to communicate with the business stakeholders, while documentation with a high level of

detail is created for (more technical) communication within the maintenance team. This shows us that different people need different LoD.

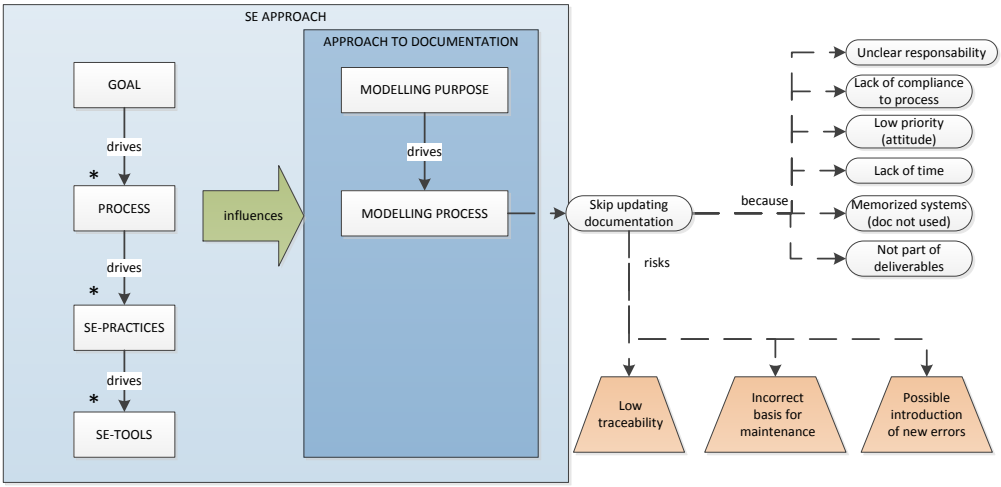

Figure 6.11. Summary of reasons for not maintaining documentation, and risks involved in not maintaining documentation.

6.4.3.2.2. Use of documentation

We also investigated how the documentation is being used. We detected first of all that the value of documentation decreases for some stakeholders when their expertise with the system increases. When a maintenance person needs to understand a system that is new to him/her, s/he first starts to read the documentation. In most cases, source code is a secondary source by which to understand the system (except in the case of a couple of developers, who prefer working directly with source code). This contrasts with the results obtained in previous research (Garousi et al., 2015), which show that maintainers prefer to refer directly to the source code itself and rely on source-code to support their information needs during maintenance tasks.We might, therefore, think that documentation is only used to 'get familiar' with the system: "*Once a maintenance engineer is familiar with a system, he hardly needs to look up information in the documentation.*" [Int18]. When a developer is not familiar with the system, the documentation is consequently perceived as useful. But once the same developer has become experienced with the system, s/he has a mental model of the system in his/her mind and uses it as a basis on which to work. Once in the latter state, documentation is perceived as less useful. These results are similar to those presented by (Garousi et al., 2013), who mention that during maintenance, documentation is used on average in only 18% of cases, while code is used with a frequency of 50%.

When focusing on the use of UML diagrams as part of the software documentation, we detected that maintainers sometimes create their own diagrams. They do this for their own use to describe the system they are maintaining and to summarise what they see in the source code. The purpose of these diagrams is not, therefore, to document

the system, and as such this type of diagrams is not stored: "*I describe a design that I plan to implement [..] and sometimes I use an enterprise architect to draw some diagrams, UML diagrams, but we don't really document a lot for development.*" [Int19]. This is in the same direction as the results of Garousi et al. (Garousi et al., 2013), who highlight that design documents do not provide that much benefit during maintenance. In this case, we detected that the design is not documented and that this would not be a problem for the maintenance tasks. However, the UML models are documented and used during this phase.

Architects can use the UML model to check that the implementation conforms to their intention. *"UML is your reference of what it should be, but you have to check if the code that is delivered is in fact aligned with your UML diagram. [..] At some point in the project, we did not do that functional test, but I also reviewed the code, to see if the structures were as I intended them to be according to the diagrams."* [Int4].

6.4.3.2.3. Usability / usefulness of documentation

We observed that the size of documentation matters. Moreover, the size of documentation has an impact on its usability. A large number of pages is considered impractical [Int18]. Some people indicate that 20 pages is already considered to be a lot. Another important factor that influences the usability of the documentation is the language in which it is written. Sometimes companies tend to force to their employees to use English as a common language in the company, yet employees would prefer to document in their native language: "*I think documentation or source code of programs should be in the language that we speak and that we can read and that we feel, and this is not English. I call it "Denglish" (Dutch-English). What another Dutch person means when he stated something in English is sometimes very confusing; then you read it and you think 'Oh, well, I thought you meant that, and not this."*" [Int36]. That freedom to use the local language would not be possible in the case of offshored projects.

It is also noteworthy that the usability of documentation is perceived differently by different people and throughout different phases of the system's development [Int26]:

- Architects prefer high-level overview pictures.
- Integration architects believe that sequence diagrams are the most important type of diagram [Int26].
- Programmers spend most of their time looking at source code. But when programmers look at diagrams, these need to include sufficient detail to allow programmers to relate the diagrams to the source code they are working with (i.e. high traceability is required).

We also detected that UML diagrams would be more helpful if they were executable models. This beats UML when that is used as box-and-lines: "*There was one project which used a BPMN*[1] *engine. With that you can model the business*

[1]BPMN stands for Business Process Model and Notation

process in the tool and actually execute it. And it was also very helpful to have another way to know how it should work, as a reference. That beats any UML diagram, because UML doesn't execute, it doesn't work, it is not physical; it is just box and lines." [Int4].

When focusing on the usefulness of the documentation, we detected that if the system (or part of the system) is simple, then no documentation is created [Int18]. Maintainers mentioned that they only document important things. We asked them next about their definition of "important things". Maintainers agreed that a part of the system should be documented if: it is a complex part, or if it is a critical part, or if the part is not obvious [Int16]. Other reasons for deciding when to document something or not are agreed by the team. Moreover, if the system's life is expected to be short, not too much information is documented.

6.4.4. Practice or style

These subsections contribute to answering RQ1.A summary of the findings obtained as regards practices or style is presented in Figure 6.13.

6.4.4.1. Diagramming practices: (standardized) UML or freeform

We asked our interviewees whether they used UML or other graphical notations during software maintenance. Some of the interviewees mentioned directly that they use UML, and others did not (Table 6.4). There were those who said that they know the UML diagrams are being used by their colleagues, and they know this because, for example, some of them have printed copies on the wall. Some others considered screenshots as graphical notations to communicate layouts on interfaces.

Table 6.4. Presence of diagrams in the documentation.

Presence of diagrams in documentation	% of interviewees
Contains diagrams	77%
UML	94%
Non UML	6%
Does not contain diagrams	23%

Almost a quarter of the people interviewed (23%) considered that the documentation they use does not contain diagrams at all, as opposed to 77% of them who considered that it does. Of those who use documentation containing diagrams, 94% of them stated that they use UML diagrams, as opposed to 6% who consider that they use other notations.

The most commonly-mentioned UML diagrams are: sequence diagrams (69%), class diagrams (55%), activity diagrams (36%), and deployment diagrams (23%). Other UML diagrams were also mentioned, but in lower proportions: i.e., collaboration diagrams, component diagrams, package diagrams, statechart diagrams and use case diagrams. These results are in line with previous surveys (Dobing and Parsons, 2006; Fernández-Sáez et al., 2015b; Hutchinson et al., 2014).

It is surprising that some interviewees referred to diagrams that are not part of the UML set of diagrams when asked about the UML set. Those most frequently mentioned are data flows and data models. The nuance aspects of notations are not used, and developers largely use the common concepts: "*I have seen so many names for the same thing. [..] UML or ERD; it is all the same, in my opinion*" [Int13]. There was also a common confusion when talking about use case diagrams, because some users considered them to be a table summarising a use case like a "diagram". This is based on the general practice of describing use cases by using a table.

We also asked UML users about the LoD of their use of UML diagrams. There were more respondents using high LoD UML diagrams rather than low LoD. Some developers also mentioned that the decision about how many details should be in a UML diagram is an architect's decision, and developers are not taken into account. This finding may indicate that there was uncertainty amongst developers as to what extent that freedom could be exercised, because a model should explain how the system works without allowing programmers too much freedom to determine implementation details, as highlighted in (Nugroho and Chaudron, 2008). On the other hand, there are developers who have access to UML diagrams (either from an architecture/design diagram or from a previous stage of development), but they do not use them because they do not have sufficient details. Developers are used to working with the source code, which contains many details, so they draw their own diagrams based on the status of the current source code in order to have a more detailed diagram, compared with the one created by the architect.

In summary: the use of graphical diagrams is very common. Within this, the use of UML is common. Moreover, the UML notation is used to represent diagrams from other design paradigms (ER, data-flow and context diagrams). Different stakeholders have a variety of purposes with models, and as a result use a range of levels of details in their models. This practice (adapting the diagram depending on the audience) is common in industry, as presented in related work (Petre, 2013).

6.4.4.2. Influence of UML usage on quality of software

We asked the interviewees about the quality of the final product and its relationship with the use of UML diagrams: "Do you think using UML has an impact on the quality of the final product? How?"

In this case, the respondents considered the quality of source code related to performing correct testing and obtaining positive results from it; i.e. obtaining a source code that is aligned with requirements and design: "*Quality is the result of checking the result too, so UML is your reference of what this should be, but you have to check if the code that is delivered is in fact aligned with your UML diagram"* [Int4]. This is in harmony with the results of the survey presented by (Nugroho and Chaudron, 2008).

Employees working on projects which do not use UML diagrams commonly believe that the presence/absence of diagrams is related to a high/low quality of documentation, respectively. It is very important to note that there is almost a general

consensus amongst all interviewees that the use of UML improves software quality (89% agreed, and 11% believed UML is not related to software quality). The reasons given in support of a relationship between use of UML and high quality are:

- The use of (automated) modelling tools improves productivity.
- The sharing of knowledge is improved.
- A peer-review process makes it easier (and highly recommendable) to adopt when UML is available.

In relation to software quality, we also asked the interviewees about the possible relationship between the use of UML diagrams and the presence of defects in the code of the system:

A couple of interviewees considered that UML usage reduces the phenomenon of defects managing to get into the code of the system, i.e. UML prevents defects, while another person replied that they believed that UML increases defects (but only whenever they are not updated). This contrast indicates that neither the positive nor the negative effects are perceived to be very significant. It can also be seen that some interviewees thought that there is no relationship between software defects and UML in itself; the defects are caused by an incorrect solution, but UML is not the problem.

More than half of the interviewees were of the opinion that the use of UML is helpful when searching for the cause of a problem in the source code. Sequence diagrams are especially valuable for this purpose: "*Sequence diagrams are also very useful, for example, if there are bugs or issues on-line and we don't know how the bug works, how the bug exists there, which components are affected and need to be looked into."* [Int11]. This is in contrast to the results of a previous survey (Nugroho and Chaudron, 2008), which shows the respondents' indecision on the impact on quality of using the UML on software testability and correctness (defect-count). Respondents in that survey might not have been exposed to testing of plans or criteria constructions using UML models, in contrast to the interviewees in this study (one of the main purposes of using UML in this study was to test guides). Early and more thorough thinking about the design leads to higher quality of design. Moreover, as stated in a quote above, while this may incur some more effort in the design stage, the respondents were convinced that there is an overall benefit in productivity. We would like to stress that although the quality of the design might be improved with the techniques mentioned in this section that does not have to be reflected in the quality of the source code in terms of an improvement. This is because the quality of source code deals with issues different to those from design, such as following naming conventions, having a correct commenting system, using a correct indentation, optimising the nesting depth of loops, etc.

6.4.4.3. Standardisation

We asked the individuals we interviewed about standardisation in their ways of working. In this case, we focussed on those standards used to document the system and the activity of diagramming. Only 1 interviewee considered that there is excessive

standardisation, while 31% believed that there is a lack of standardisation. These respondents felt a need for more standardisation in relation to the following:

- **Naming**: naming conventions for classes, attributes, etc., in code and diagrams.
- **Layering**: it is not clear what the recommended layering of the system is.
- **Style**: There are many issues related to the style of diagramming (and subsequently of coding) which are not clear.
- **Level of detail**: it is not clear at what level of detail systems should be modelled.

"*We use different terminology, different naming, different layering*" [Int1]. They argued that the standardisation should be established in the early phase of a software life cycle, in order to obtain the maximum benefits during the rest of the project [Int14]. They also emphasise that the standardisation should be done right across all the teams if it is to be successful [Int4]. The interviewees considered that standardisation plays a very important role when third parties are involved [Int12].

Although the needs mentioned above arise when asking for diagramming, some respondents highlighted the need to also standardise the source code (naming conventions are needed) [Int17], and text (what should be written and the structure of the document) [Int25]. The main benefit of introducing standardisation in any of the items of the software documentation that was mentioned is always a reduction of the risk of misunderstanding [Int23]: "*A standard is something that will be understood by all people, across all platforms.*" [Int28], or "*The problem in ICT is that every platform uses its own language, its own way of looking.*" [Int28].

Independently of their opinion on the presence of standards at the company, many of those surveyed agreed that there is a lack of compliance with the standards. They justify this, using the following reasons:

- The way they work is intuitive, and they do not need to follow other standards [Int10].
- They have previous knowledge, and they do not need to follow all the standardised processes [Int10].
- They believe that standards do not help in every case, and sometimes these standards are not applicable [Int16].
- They also believe that there are items, such as text which, unlike diagrams, cannot be standardised [Int28].

Those who mentioned that they do not follow standards explained how they work. The majority of them use solely box-and-line diagrams (19%), while others use standard UML, but in a non-strict manner: "*Because there is no real standard, people use their own inventions*" [Int18]. This finding is aligned with "selective use" of UML detected in previous studies (Petre, 2013), where UML is used in design in a personal, selective, and informal way, for as long as it is considered useful, after which it is discarded. Advanced features of the notation are not used, and do not therefore need to be updated. Moreover, some interviewees use diagrams that are similar to UML, but may not use all its elements correctly (for example the different types of arrows in a

class diagram) [Int3]. In some cases they use the formal UML specifications, but they mix them with their own notation to complement them [Int5], and they complement the diagrams with personalised legends in order to clarify the non-UML part of the diagram [Int23].

The individuals who gave their opinions also noticed a problem with standardisation: when a system is very old (especially in the case of legacy systems), the documentation contains a mix of different standards that had been adopted for several years. We are therefore of the opinion that standards have a short life-time: "*I've heard a lot of standards over the last 13 years, and what is standard now isn't so standard in few years' time, because then there is another standard. [..]What is standard now is not so standard 5 years from now*." [Int27]. One benefit of using only 'box-and-lines' in diagrams is that it makes them independent of the evolution of the UML notation.

It was also surprising that maintainers know that the diagram they are creating is not correctly written, but they deliberately write it in that way in order to trigger discussion: "*It is better to be unclear than to be misunderstood*." [Int12]. The idea behind this is that a reader of documentation will recognise that a part is unclear, and this triggers him or her to ask for clarification. Whereas a misunderstanding will lead to a wrong (design) decision.

In many cases (63%), Visio is used to create diagrams. Visio supports the UML notation, but does not enforce the syntax as strictly as a dedicated UML-CASE tool does. Arbitrary graphical shapes can be connected without regard to their meaning and hence without keeping syntactical rules. Moreover, Visio does not create an actual model of the system, only a diagram. Visio thus does not support consistency inside diagrams/models or across diagrams/models.

It was also surprising that in some cases, the interviewees are following a standard notation but they do not know that they are. They merely copy the way of modelling that is already being used in the documentation: "*if sometimes I have to make new diagrams or I have to change them, what I do is just use what is used in that system*" [Int27]. This means that the formalised documentation is sometimes produced using previous documents as a basis.

Another problem with standardisation is that it is sometimes very strict, and slows the maintenance process down. There is usually a formal procedure for the approval of milestone-documents. This process typically takes a long time, and thus slows down the development. This discourages people from asking for approval or even from making (small) updates to milestone-documents [Int18].

Mechanisms to incentivise the correct use of standards should thus be introduced: "*If you let people choose, you lose all your advantages. So, yes, force them*" [Int8].

Sometimes standardisation is not provided by the company, and producers of documentation agree with the consumers of the documentation on the meaning of their diagrams (it would appear that this occurs in a just-in-time-manner). Once common

agreement is in place, this is an effective way in which to work: "*Because there is no real standard, people use their own inventions*" [Int18]. The only risk to this is a change of staff. This practice of using a common agreement about meaning takes place because the team members, and especially the business stakeholders, sometimes lack knowledge [Int23].

We found that there are also some concerns regarding the use of documentation as a standard step of the maintenance process that are related to the team members' attitude.

- Some developers do not like documenting, and they consider that this is the responsibility of other team members: "I am not a writing person, I am a building person. I hate documentation because I am a technical guy. [..] We are not documentalists. We are builders. That is a different state of mind." [Int13], or "Developers don't like to write documentation" [Int6].
- The documenting part of the project is considered to be a boring part, and maintainers try to avoid it: "documentation is almost a dirty word you should not use." [Int35].
- From a selfish point of view, maintainers believe that documenting is work done only for others "If I write documentation then I am helping a colleague." [Int13]. Previous research (Lutters and Seaman, 2007) has highlighted that documentation that is written from the perspective of a maintainer (and is sometimes even written by a maintainer) is especially useful. It would, therefore, appear to be important to take this point of view into account.
- Sometimes developers do not follow the specification provided; they do the maintenance and then implement what they consider to be a better solution: "often they [developers] just don´t listen. They just do it the way they think they should do it." [Int1].

A summary of the findings about "PRACTICE" discussed up until this point is given in Figure 6.12.

6.4.4.4. Use of reverse engineering

During the interviews, we asked the subjects about the use of reverse engineering, especially when diagrams are not available. The first observation is that sometimes people do not know that a (UML-case) tool can help them to automatically partially extract a diagram which can be used as a basis for creating UML diagrams.

In the case of projects for which UML diagrams are available, the diagrams (along with other documentation) are in many cases not updated according to the changes in the source code. In these cases, the interviewees considered the use of a reverse engineering technique to be a very helpful tool in obtaining a trustworthy diagram. We found that reverse engineering sometimes takes place, but this process often requires additional manual work. Indeed, the interviewees considered reverse engineering to be a lot of work: "*That's a nasty question, because the documentation should preferably be up to date. You want the documentation to be correct, so try to get documentation*

up and running up to the situation as it is. [..] In practice this is a very tedious job, because you have to be really specific if you want to describe a piece of software. It basically means digging into the code and seeing what is going on exactly here, and here. As I'm not a Java expert this really delays the process, because you need both kinds of people to reverse engineer the documentation. That means it is usually not done" [Int14].

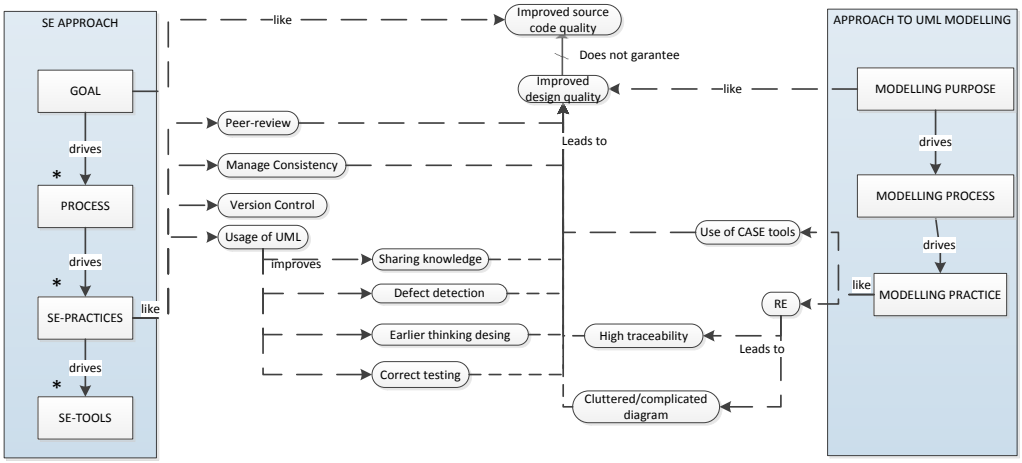

Figure 6.12. Summary of findings about Practice.

Alternatively, when no graphical documentation is available, UML diagrams are manually RE to attain an understanding of the source code [Int19]. We therefore found some cases in which people carry out automatic reverse engineering, but after the automatic process, a manual process is needed, when the architect tries to "clean" the diagrams. The people who responded considered this process to be difficult, because they were not able to deduce the implicit policies that were used to generate that specific diagram, or why something is structured in that way, or if a design pattern was used: "*You have to go through the code in a rather labour intensive way and look at the classes most of the time. You have to try to deduce, on the one hand, the business concepts and the patterns that have been applied, and the policies that have been driving the design, how clear any policies were, and then you make the diagrams. Of course you can do automated reverse engineering of all classes that have been implemented. But this would also produce the framework classes, so the level of detail is extremely high if you do automatically. It is actually useless to do it like that. You therefore have to go through the code as a human being and try to deduce the intelligence that is in the design, and then make the model. In that sense it is less useful than if you have applied it from the top-down approach*" [Int1]. In reverse engineering "*it is hard to understand why a certain function was there, or is not there at the present time*" [Int24].

Although the "cleaning" process is carried out by the architect in the early phases of the maintenance project, there are developers who would like to have the original

reversed engineering UML diagrams, because these are more traceable to source code. In such cases, they would consider the UML as the "truth". However, this approach loses followers when the projects are very big and diagrams become unreadable.

We asked the UML users if they would prefer a diagram which originates from a forward design process or from a reverse engineering process. In most of the cases (70%), they prefer the forward design processes.

Some other problems found with Reverse Engineering are listed below:

- Models capture information that cannot be extracted from the code. This is an architect's perspective: "*Reverse engineered models are completely not useful. Too detailed. Not the right semantics. No abstraction.*" [Int4].
- Sometimes the RE diagram is drawn up by a single developer; the design is not therefore discussed and hence not disseminated between multiple developers in the team. Reverse engineering eliminates the growth of a shared understanding [Int4].
- Programmer's perspective: "*I prefer the reverse engineered diagram over the diagram from the architect because it is easier to update with the changes that I make to the code. […] To get an overview I don't recommend using RE, except when there are no forward diagrams.*" [Int11].
- FD and RE design models may use different naming for class names, methods and operations. As a result, it is not easy to use them together/merge them: "*reverse engineered diagrams are a bigger truth than the forward engineered diagrams, because … you may have different names of objects in the design and in the code, but in reverse engineering there is no choice, you just import your code, so there is no way you can have this.*" [Int37].
- RE diagrams contain too many details: "*there are properties of an object, which I will never model; I just say it has a list of properties. Then in a reverse engineered model, I get like 10 properties actually, and so I have the problem of 'Oh what is this anyway?'*" [Int24]. "*The detail of reverse engineered diagrams is not very usable*" [Int3]. "*For projects with lots of code you get lots of diagrams, with all the details, but they are not really readable. [..] If these tools generated some diagrams with less details, that you could specify how detailed it should be; then maybe they would be useful.*" [Int19].
- RE diagrams do not recreate the desired layouts: "*I could reverse engineer from an SAD to UML, but then the whole thing is: it has to be rearranged*" [Int3]. The layout created in forward designs contains semantically-meaningful information for the designer. For example, related classes are close together (even if there is no association between them).
- Reverse Engineering is time consuming: "*it took me ages to do reverse engineering and then figure out those sequence diagrams.*" [Int37]. This relates particularly to extracting/abstracting dynamic information.

In summary, we conclude that:

- For programmers, reverse engineering may be a practical way to obtain a diagram that describes a fragment of interest of the system in hand. Yet programmers prefer the design to be updated automatically.
- For architects/designers, reverse engineering is not considered a practical option because:
  - It yields diagrams that are too detailed and that require significant efforts in manual processing. In particular, details need to be left out, and a meaningful layout needs to be created.
  - The Reverse Engineering functions of current UML case tools fail to recreate dynamic/behavioural views of the systems.
  - Using Reverse Engineering to recreate a design model after the facts (i.e. after (much) of the programming has been done) may result in the omission of discussion about the design that would otherwise increase the shared understanding of the design of the systems amongst the developers.

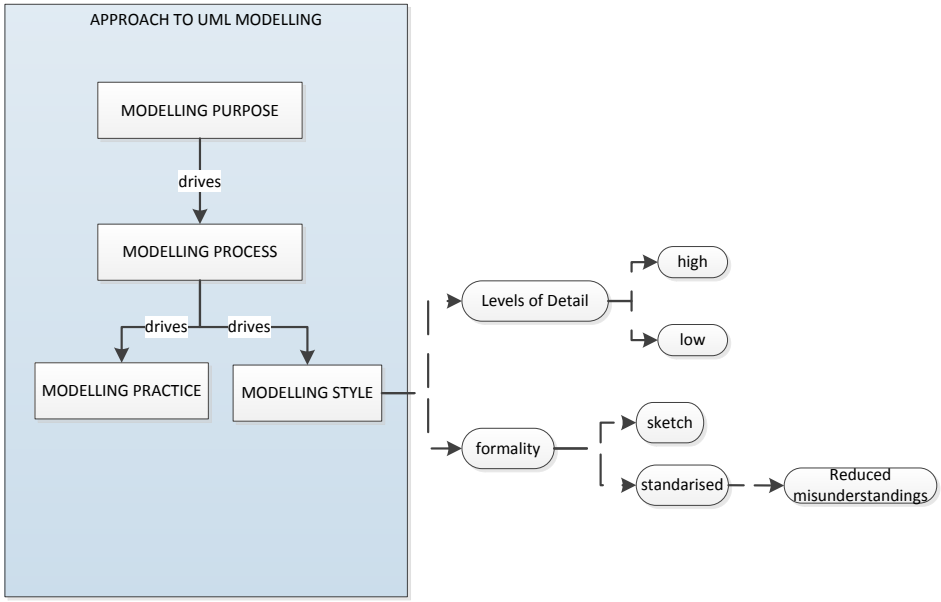

Figure 6.13. Summary of findings about Style.

6.4.4.5. UML versus other graphical notations

We asked the interviewees to compare UML with other notations which are in use at the company and which might be used instead of UML.

Some of them referred to Archimate as an alternative notation to help in system modelling. They were of the opinion that Archimate is a better notation for business models [Int2] than UML, but that UML is more useful for modelling functional documentation. UML is also considered to be a modelling language that is easier to discuss (especially for developers) than Archimate [Int26]. One advantage of using Archimate rather than UML is that Archimate comes with some guidelines for design,

such as layering to help in design [Int1]. The disadvantages cited by the interviewees are that Archimate is less powerful, and too abstract [Int23]. Nonetheless, one person considered that the use of UML in combination with Archimate would be very beneficial [Int1]: "*UML is easier, so I would say in discussing, while designing or building systems, or in making the preliminary design steps it is better to have UML, because it's easier to discuss. But if you are working on the border-line between say, hardware, system software and software, ARCHIMATE is better, I think, because it can warn you of certain problems that might arise if, for instance, you design a heavy application and have only one server. So it depends on your goal*." [Int23].

Other notations were also mentioned as good candidates for complementing UML:

- User-interface perspective/Screen layouts [Int18] and maybe screen-flows.
- Executable specifications (esp. BPMN) [Int1].

Finally, Domain Specific Languages (DSLs) were also referred to. The diagrams generated with this kind of notation are not regarded as adaptable as UML, because they are only able to present one dimension/view of the system. This would have the advantage of providing a better understanding of that view of the system, because it might contain lots of details. But there is the risk that something else might be lacking, because diagrams are usually created from one point of view (structural view, network view, behavioural view, etc.). All the aspects may not be placed together: "*But sometimes our projects deliver those messages in DSL itself, which I find so much easier to look at, because you can see it more or less in only one dimension; it's a very hierarchal thing. In UML you can put things next to each other; you have 2 dimensions.*" [Int3].

When we asked the interviewees about the modelling notation, some of them were not able to answer which notation they were using. When digging deeper into this category we discovered that 38% of the interviewees uses arbitrary graphical notations (or box-and-lines), or that only the basic elements of the UML syntactical notation are used in documentation: "*Rectangles and lines. It does not have any symbol [..] with arrows, dots or hollow or full circles.*" [Int18]. Some of the reasons for following this approach are summarised in the following list:

- Some developers *adapt* their notation to the one that is already used in the existing documentation. Being consistent with one existing document is of more importance than having consistency in all the documentation for different systems. The documentation of old systems, therefore, typically uses old graphical notations, and continues to do so even after recent extensions.
- Using only 'box-and-lines' in diagrams makes diagrams independent from the evolution of the UML notation. If advanced features of the UML notation are not used, then they do not need to be updated either. The fact that the UML notation has had several major and minor revisions of its standard (at least 3 major ones in 10 years), has been a factor in reducing the eagerness to conform to all the details of the syntax.

- People that create documentation assume that the reader has a certain level of domain knowledge [Int21], but sometimes this is not entirely true, because the consumer of a document is, on occasions, a business stakeholder.
- "*You can achieve the same with Visio as with UML, but you need to align how to use it.*" [Int26].

We researched the conformance of diagrams to the UML standard(s) in a collection of 35 Global Software Designs at our case study company. This set shows that 40-50% of the diagrams are formal UML diagrams, compared to 60-50% of the diagrams which are not UML. In this last group we found some diagrams of other notations, or diagrams which seem to be UML but which are actually not (box-and-lines, or people's own inventions).

6.4.4.6. Comparing text and diagrams

One important issue that was discussed during the interviews was whether the use of diagrams (especially UML diagrams) is more helpful in understanding the system than documentation that consists only of text. Those interviewees who had been working at the company for a very long time, who had not used diagrams during all their time at the company, and who had not had training on UML diagrams, considered that diagrams would not be helpful for them. We might explain this as being due to fear of change, but also because of unfamiliarity with the notation. In contrast to this group, the majority of those who replied to the questions considered diagrams to be a very helpful tool as regards understanding the systems and the changes which need to be made to them. They based the benefits of diagramming on the possibility of increasing the level of abstraction of representing the system. The following sentence was repeated in the majority of the interviews: "*A picture explains more than a thousand words.*"

The interviewees who argued in favour of using diagrams because they believe that this improves the understanding of the system, did so for the following reasons:

- People are visually oriented [Int5], and they usually prefer visual notations [Int24].
- Diagrams help when searching for something. It is faster to look for something in a picture than to read a long text, because of navigational issues. This view is supported by the majority of interviewees, although one person did not agree, pointing to the fact that there are lots of tools that support text search [Int37]. A reader can judge quickly whether a diagram contains the information s/he is looking for. Judging this in text requires paying closer attention and thus more time: "*People are more likely to skip over a piece of text than over a diagram.*" [Int31].
- Text is so much more difficult to maintain than diagrams [Int37]. This may be related to the previous bullet: it is difficult to find the piece of documentation to maintain. Furthermore, it is important to highlight that the maintenance of diagrams requires skilled people (the maintainer, at least, needs to know the notation), but the maintenance of text might well be done by anybody [Int25].

- Diagrams are easier to understand than text [Int32], although depending on the type of diagram, not everybody might give the same interpretation [Int28]. In relation to the understanding category, we detected that pictures solve problems of understanding for dyslexic people. This is not a very common condition, but it was one that some of the interviewees suffered from [Int13]. Some interviewees also highlighted that diagrams are easier to understand in the context of systems because they describe relations in easier simpler way [Int12]. Moreover, the presence of diagrams in legacy software documentation is very valuable for an easier understanding of the system [Int19].
- Diagrams are easier to compare than text: *"It's easier to put two diagrams next to each other and look at them and see the relation between them. Text?... it's much harder in that."* [Int12]. This means it is easier to detect what has changed in a diagram than in a text.
- It can also be said that text presents difficulties as regards facilitating the traceability with other documents, or with source code [Int9], because diagrams are better at representing structures: "*text just lacks the means of connecting all the components, and it's also very difficult to keep track of all those dependencies that you have in the environment when somebody is just telling you what there is/are, or is writing down everything simply because you want a quick overview of everything that there is*" [Int31].

Some of those surveyed said that text is regarded as cheaper (in terms of cost) than diagrams [Int28], because everybody knows how to write, but not everybody knows how to use a modelling notation. Modelling can enforce systematic use (syntactic correctness), but when using natural language text, much more freedom will be used in writing [Int28]. There is also the issue of information density. Diagrams can convey a lot of information. This is good for abstracting, but may be easily misunderstood by non-experts in the notation [Int12]. Furthermore, creating graphical representations early during development enables such errors to be found and corrected while their repair is still 'relatively cheap' (this is in comparison to refactoring such structures later in the project when much more code will need to be checked/changed). The lack of a systematic use of language might lead to more possibilities of errors creeping into text than into diagrams, because creating a diagram entails a more detailed thinking process: "*I think you can make a mistake more easily in text than in a diagram, because in a diagram, you have to think more about... While you draw, you can see the mistakes you made in the text, because things don't match anymore. They can be going in a totally different direction...it's impossible to draw this when you try to put it down. I think it (*a diagram*) can help.*" [Int10]. In relation to the presence of mistakes in text or diagrams, the interviewees considered that mistakes in diagrams lead to greater errors than mistakes in text [Int27]. If a diagram is not correct, then this is a big error. This may be because information that is covered by diagrams relates to architecturally-significant aspects of the system: "*People take [a diagram] as the truth sooner than a piece of text, so they might be less critical towards it*" [Int12].

The interviewees also considered it necessary to use text to explain diagrams, as a complement to the graphical information, because diagrams alone are as helpful as text alone: "*a picture is mostly saying more than the text alone, but it's the combination, because you have to explain what you want, and that's done in text*" [Int10]. Text usually is too detailed in comparison to diagrams and an overview cannot be extracted [Int9]. Diagrams provide a global, high level overview, while text explains rationale and provides details [Int24]. This means that text is considered better for explaining details [Int12]. "*Your first thoughts really show where things are changing and different, and highlight the aspect that you need to address. That needs an explanation, because it's not the final picture*." [Int3]. Text is therefore used to explain/highlight changes, differences from previous designs, and also exceptions in flows [Int34]. We would also like to highlight that the complementary relation of diagrams and text could be extrapolated to the context of oral communication, where talking provides extra information to complement written documentation: "*Talking to people can give info about the history*." [Int12], and "*When I explain a diagram, I emphasize changes, differences*." [Int3].

6.4.5. Tooling

In the SE community' discussion regarding the adoption and effective use of modelling, the category of tooling was identified by several researchers (e.g. (Whittle et al., 2013)). In this section, we summarise our main finding that relates to the tool support for modelling and documentation discovered through the interviews. These results contribute to answering RQ3.

Several participants in our case study indicated that they like to mix text with UML diagrams, and also with informal box-and-lines drawings. In reality, virtually all software (design) documents are indeed a mixture of text and diagrams. Unfortunately, current UML CASE tools do not provide any support for this (Chaudron and Jolak, 2015). From a complementary angle, word processors do not support, but rather hinder, the updating of diagrams. We conclude that developers would like tools that allow them to flexibly add text and custom notation to UML diagrams. Yet even today – 20 years after the introduction of UML - such tools do not exist. The need for tools to support the mixing of text and formal diagrams and sketches has been supported by other studies (Dekel and Herbsleb, 2007).

There are several tools that support the management of different versions of textual documentation or source code. But in the case of UML diagrams, the support for versioning is immature, especially in the case of the merging and diffing of models "[when asking about UML's disadvantages] *that we don't have real versioning on the UML design has more to do with the tool that we use; it's not related to UML. I think it has more to do to the tool*." [Int26], [Int29]. Sometimes this lack of tooling is substituted by a definition of a human process: "*We have the live structure and development structure. So basically what will happen is, if a document needs to be updated you take a copy of the live version, make the changes so that it becomes the development version, put a double tag saying it's been updated and give the dates it's*

been updated and stuff, and say what's been updated. And once the codes been done and loaded, it can be moved from there, and then it's an algorithm." [Int20]. When working in this way, there is a new task which should be done by a team member, which could be done automatically if there were a tool with which to manage the diagram versioning.

The same lack of tooling applies when the maintainer aims to reuse a model or a part of it. It would appear that UML tools do not support any notion of modularity of the model [Int8]. As a result, it becomes difficult to reuse parts of a model.

There is no specific notation support in UML as regards representing design decisions/design rationale ('why's') or linking these to the actual design. As stated in (Aseniero et al., 2015; Burge et al., 2008; Kruchten et al., 2009), design rationale should be documented to facilitate understanding and maintenance. The explanations of designs currently need to be written in a separate document. There are two key problems when following this approach:

- The high likelihood of losing the traceability between model and text, thus leaving the documentation and model out-of-synch.
- The 'Dizzying' of maintainers when they are reading the documentation: "*I hate jumping between documents; that makes it hard to have an overview of what's what.*" [Int12].

In relation to this, a common problem of UML tools also arises; there is poor support as regards searching in models.

We also found that if people obtain training in modelling, then this is targeted towards understanding and using the notation, but not towards the use of the tool. Given that the effective use of UML requires the use of advanced CASE tools, it may be wise to also invest in training people how to use such tools, showing their main features but also their details. Some people that regard the tools as intuitive are against this recommendation.

A majority of those surveyed (12 people) state that the use of a tool would help them to correctly model a system, especially because tools help to create syntactically correct diagrams [Int14]. Conversely, one interviewee mentioned that using a tool to model a system is a waste of time; he prefers to create a model on paper/whiteboard and take a picture of it to then attach this to the corresponding document [Int16]. But it is clearly difficult to maintain a model (keep it up to date) in this manner.

The needs of the modelling tools which were highlighted by the interviewees of this case study are summarised in Figure 6.14. These needs complement the list of desirable features of a proper tool for software modelling presented by Forward et al. (Forward and Lethbridge, 2002), who mentioned that it would be useful to have tools with which to track changes in a software system for the purpose of updating and maintaining its supporting documentation. According to the results of Forward et al. (Forward and Lethbridge, 2002), in order to track changes between documents and source code, the technology must be able to relate:

- documents to source code,
- source code to documents,
- documents to other documents (in, for example, hyperlinked environments).

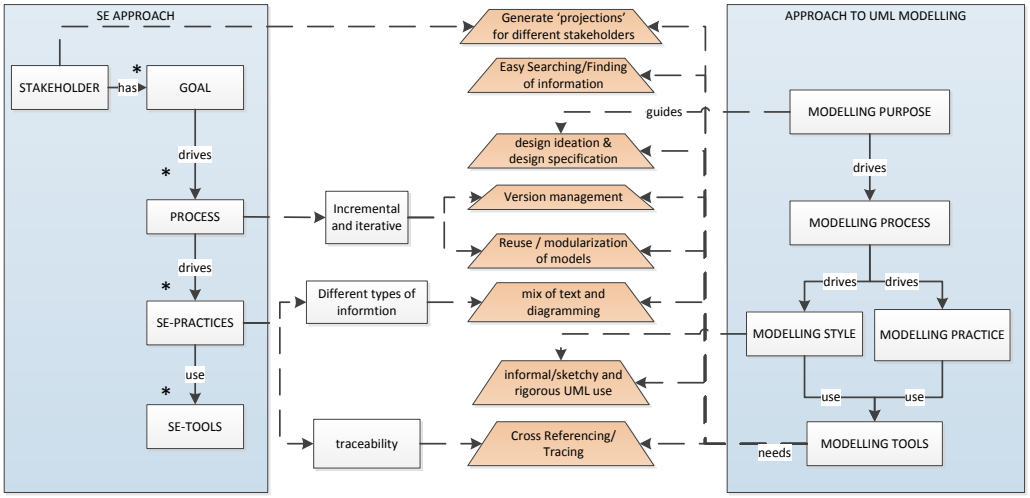

Figure 6.14. Needs of modelling tools detected in the case study.

6.4.6. Context

When able to fit the question into the interviews, we asked about factors that would influence the use of UML in a project. By sampling the respondents, we found that the larger size and complexity of a system are factors that increase the likelihood of UML use.

We also detected some details about the influence on the approach to documentation and modelling of two kinds of maintenance projects which have special requirements. They are the outsourced maintenance projects or offshored maintenance teams. These subsections contribute to answering RQ3.

6.4.6.1. UML and outsourcing/offshoring

Several of our interviewees had worked on maintenance projects which involved outsourcing or offshoring (4 people, to be precise). In those cases, the respondents mentioned a special importance of documentation (especially documentation containing UML diagrams), based on the following reasons:

- To improve communication (mentioned by 3 out of 4 maintainers enrolled on outsourced/offshored projects): "*Especially for offshoring teams it is very important that your documentation is correct.*" [Int5].
- To guide the implementation: "*They are doing the programming based on that notation (UML).*" [Int5].

- To check the implementation (conformance to design): "*UML is your reference of what it should be, but you have to check if the code that is delivered is in fact aligned with your UML diagram.*" [Int4].

A lack of modelling also influences projects in which multiple parties are involved. This deficiency leads to a corresponding lack of clarity in specifications, which aim to precisely define the work that is demanded from a subcontractor, or to describe an agreement on the scoping of responsibilities between teams.

If we instantiate the element "CONTEXT" of the baseline theory to present the results obtained about those maintenance projects that have an outsourced team or in which the maintenance is offshored, the findings on this subsection are summarised in Figure 6.15.

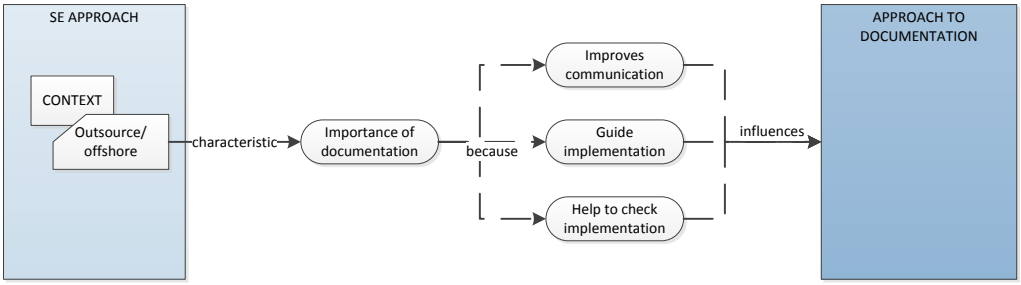

Figure 6.15. Influence of outsource/offshored maintenance on the documentation approach.

6.4.6.2. Legacy documentation and modelling

We investigated the presence of graphical notations on the documentation of legacy systems. There are a large number of systems that were developed in the pre-UML era. The documentation of these systems does not contain models. The older the system is, the fewer diagrams it contains (also driven by the fact that old computer systems did not support graphical user interfaces). Migrating this documentation to make it compliant with present-day documentation and modelling practices is a huge effort, and hence a huge investment [Int8]. No automated tooling seems to be available for this, either because reverse engineering does not produce the right abstraction, layout and behaviour models, or because there are no reverse engineering tools for old programming languages.

Moreover, for many legacy systems, there is a discussion about when to phase them out (end-of-life) and replace them with new systems. In such cases, companies are even more reluctant to invest in the documentation of these systems, while this documentation could, at the same time, be very valuable as regards building the replacement systems. It should also be said that in some cases the system is expected to be replaced by a new version soon, which means that no more investment is made in that legacy system; sometimes the replacement never occurs, however.

There are occasions during maintenance engineering on legacy systems when developers create diagrams (e.g. for their own understanding or to plan a solution), but

then there are no clear incentives for using these diagrams to update the documentation. Moreover, if diagrams are made, they are often in the same style as found in existing documentation, and hence use box-and-lines or older diagramming notations (dataflow diagrams). What is more, screenshots may find their way into documentation.

Furthermore, some people working on legacy systems are not trained in new notations like UML. So they cannot create UML diagrams. In addition, young people (with knowledge in UML) who are added to legacy projects are constrained to use a notation that all team members can understand.

If we instantiate the element "CONTEXT" of the baseline theory to present the results obtained about those maintenance projects of legacy systems, the findings on this subsection are summarised in Figure 6.16.

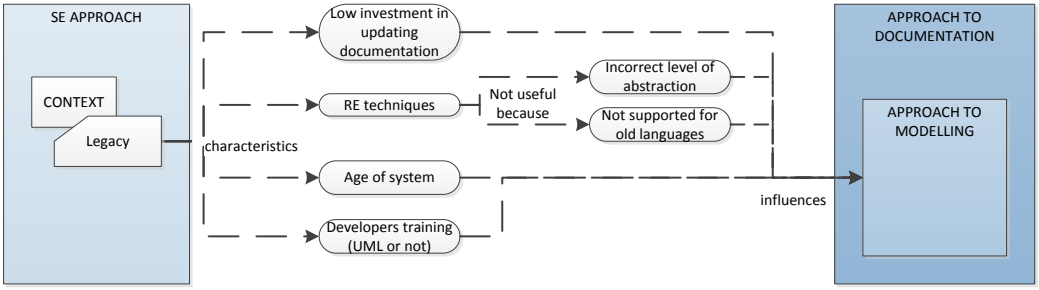

Figure 6.16. Influence of legacy systems on the modelling approach.

6.4.7. Other Findings

One interviewee had recently started working in an agile team after spending 15 years following a rigorous waterfall approach in the mainframe area. In that style, documentation had to be complete and signed off before it was handed over to the next phase. He is new to UML and agile development: " *... and I say this carefully because I'm new in this department, but I have noticed there is a lot of fixing smaller errors after implementations too, and this is definitely different from what I was used to; when it came to the time to implement [..] anything that was wrong made the alarm bells ring l. [..] Now it seems like it's more or less acceptable to a certain extent to be less strict on this.... or maybe it's just the way the e-services are connected or how different teams are working, so the chances for such defects or errors is bigger.*" [Int35].

Furthermore, during the analysis of the interviews we detected some misperceptions that we consider worth mentioning:

- Tooling is expensive: We made an inventory of the tools in use at the company: Visio (15% of people using a modelling tool), Bizz Design Architect (5%) and Sparx Enterprise Architect (80%), taking into account that one person might use more than one tool. The prices of licenses for these tools are between 135€ and 160€; a total of 150 licenses were needed in an ICT department of 800-1000

employees. In addition, between 4,000€ and 6,500€ per year was paid as maintenance costs related to the use of the tools. For this size of an ICT department, the costs for tools are relatively very small compared to the yearly budget (mostly in manpower) of software maintenance projects. Moreover, the costs of tooling are fixed costs which can be paid off fast. Tooling costs were not considered an issue by the management of the ICT department. The perception of tools as an important cost is also highlighted in the survey performed by (Hutchinson et al., 2014). That survey focused on MDE, but the main tool used for modelling at the Company, Enterprise Architect, is a popular tool for MDE.

- An often heard argument in favour of not prioritising the maintenance of documentation is that 'the business' stakeholders or clients do not value documentation. While we agree with this, we believe that the use of modelling and updating of documentation is a technique that is internal to the software developers and that can be used to produce software faster and of higher quality. The business stakeholder only sees the shorter development time and higher quality. The same reasoning applies to the following set of arguments: "Business stakeholder does not understand the diagrams. Business stakeholder does not see the value. Business stakeholder does not want to pay for this". Instead, we would recommend reinforcing the idea that models are for internal use to run the project more efficiently. This contradiction (business view vs. development view) may be linked to a short term vs. long term trade-off: not making documentation at this time may lead to slower and more expensive maintenance on the same system at some future time.
- UML is identified with the RUP-or Waterfall-approach. It is a fact that the RUP is a methodology that is based mainly on UML, and that Waterfall is usually taught using examples of a methodology to introduce UML. But the use of UML is not limited to these two software development methodologies. UML can be successfully integrated into other software development methodologies, like agile methodologies.
- Model-Based Systems Engineering (MBSE) is a technical approach, and hence not a solution for team or organisational units that are not working well together. In general, technological improvements only bear fruit when the team is working well together.
- It was not clear which person/role is responsible for maintaining the documentation. All the interviewees tended to fix their attention on their colleagues to establish which person was responsible for that purpose. The person ultimately responsible was the project manager, but it would appear that he did not state who was responsible for this maintenance task.

Finally, we detected gaps between the interviewees' opinions that should be highlighted. There are three issues here:

- There is a gap between the architect and the developer. While architects spend a lot of time creating precise and detailed diagrams because they believe that developers

like to use diagrams, some developers prefer to work directly with the source code. But the problem is not based on the notation, because developers also generate their own UML diagrams and introduce more details. The problem of the architect's diagrams might be related to the solution reflected in the diagram (constructed without the final developers' opinion), or to the low traceability of the diagram with the source code.

- There is also a problem related to the nature of the diagram. The developers believed that diagrams should be mainly a representation of the technical layer. But some architects have a tendency to move towards business orientation (modelling business process and business concepts). Furthermore, in the business, representatives interpret UML diagrams as technical pictures.
- There is a big gap too between the source code of a system and its documentation. This is based on the fact that when time pressure appears in a software (maintenance) project (which is very common) one of the first things that is sacrificed is the documentation. So, at some point, the documentation becomes out of date and no longer represents the system, which means that maintainers can no longer trust the documentation.

6.5. RECOMMENDATIONS

Using the results of the case study presented in this paper as a basis, we would like to provide the people involved in modelling and documenting practices with some recommendations. Some of them are already being considered by the interviewees and they are mentioned as best practices; others are "wishes". We summarise those recommendations that we considered to be most noteworthy in the following list, organised by categories. This list of recommendations contributes to answering the RQ4.

6.5.1. Purpose of use

- Reflect on and define the purpose of using UML [Int24], and also define which diagrams are going to be used for each purpose.
- Involve developers in the software design process [Int4].
- Do not throw documentation 'over the wall', but have interactive (face-to-face) meetings where the models are explained and questions can be asked about it. The absence of questions is more likely to be an indication that people did not look at your diagrams. By looking at the results produced by 'consumers' of the documentation you can detect whether there are any gaps in the documentation [Int4].
- The architect should send the model to the team, particularly in offshored projects. That team should then review this and provide the architect with feedback to be checked (all using UML) [Int4].

6.5.2. Processes

- Ensure that the documentation gets updated:
  - Plan for the creation and updating of the documentation [Int11]. Reserve time in the project to update documentation [Int11, Int19.
  - Updating models/and documentation should be part of your 'definition of done' [Int24].
  - Provide clear criteria that define who updates, as well as when and how documentation should be updated:
    - "The moment you change the structure, you should update the diagram" [Int26]. Or at least, maintain the documentation every week/month [Int4].
    - Define the person/role responsible for maintaining the documentation.
  - Keep a to-do list (backlog) of documentation updates that need to be performed [Int37].
  - Establish a way by which all stakeholders of a project can quickly and easily determine whether a design/document is up-to-date.
    - Establishing a method of versioning may be part of a solution to this.
- With regard to the team size and its relation to documentation:
  - The use of modelling and documentation becomes more useful in the case of teams of 6-8 people or more [Int13, Int28].
    - Documentation is more useful for projects that last longer than 1year.
- Decide on a maintenance/migration policy for models (and documentation) [Int24]. When migrating legacy software, pay particular attention to when diagrams/models are introduced. Special guidelines may be useful to standardise which diagrams to introduce and, if making separate additions to existing documentation, where new types of documents are stored in repositories/file systems. [Int18]. Moreover, legacy documentation may not be well structured, and a significant effort may therefore be needed to arrange available information into structures/templates of the target documentation standard [Int18].
- Finally, in order to improve quality assurance, we recommend the integration of peer-reviews for models and documentation in the development process[Int6]

6.5.3. Training

- Invest in training people on how to use the modelling tools, showing not only their main features but also their details, given that the effective use of UML requires the employment of advanced CASE tools.
- We recommend giving incentives to those software engineers who are not familiar with these advantages and who do not know any possible positive aspect of a change towards using modelling in their process, so that they can become aware of the long-term benefits of using modelling languages (especially UML). Not only this, there is a need for training in that sense. Software engineers should also be encouraged to realise the benefits of maintaining the documentation, and/or be trained in taking advantage of those benefits.

- Teach UML separately from methodology (like RUP). Sometimes it is not easy to apply UML correctly in other methodologies when you learnt the notation as part of a specific methodology. Moreover, some drawbacks of RUP may be projected onto the use of UML.

6.5.4. Standardisation and governance

There is a need for standardisation, which should focus particularly on providing standards/guidelines related to the style of modelling and its archiving:

- Archiving (organisation of files and naming of files).
  - If documents are not used for a long time, people forget where to find them [Int18], and flexible searching mechanisms are therefore important.
- Conventions for naming models, classes, methods and attributes should be defined.
- Define design conventions: which design patterns/strategies are to be used in which situation. The scoping of components and layers should also be established. This helps ensure that similar problems are solved using the same solutions – thus achieving architectural integrity/uniformity.
- Complement the diagrams with personalised legends in order to clarify the non-UML or non-standard parts of the diagram.
- Define the level of detail which should be presented on diagrams. This would help to avoid an excess of documentation, or a lack of it.

It might be helpful to ensure that producers and consumers of documentation know each other. This would thus enable them to agree on standardising the aforementioned issues, and also agree on the relevant information to be documented and the level of abstraction which is needed in each case. Moreover, it will increase compliance if people believe they are 'helping a colleague' [Int13].

Alignment of vocabulary is also necessary, but this is independent of the use of UML [Int4]. Yet at the same time, because UML is a standardized notation, UML can help in the definition of the common vocabulary.

Some standardisation is also needed at project and organisational level in order to solve problems related to the accessibility of documentation, always using the same project structures (for example in directories and documents naming, typed of files, etc.). It would also be helpful to use standard templates for documents [Int18].

6.5.5. Tooling

- Make modelling tools available from the start of the project, in order to obtain their benefits from early stages in a project.
- Use of tooling for automated checking of UML, e.g. consistent use of patterns, design principles, naming conventions, or correspondence of source code to design. These may be applicable for long-lived projects that have a high maturity (in order to increase the completeness and correctness of the documentation).
- Use tools that support searches in models.

- Finance tooling centrally – not at the project level. This prevents projects from trying to circumvent modelling by avoiding (generally small) tooling costs.
- Use tools with functionalities related to the traceability between model and text so as not to leave the documentation and model out-of-synch.
- A very important issue which should be improved is the need to keep diagrams and the documentation in-synch with source code, representing all the changes made to the system in them. In order to keep the diagrams updated, we recommend the use of a version management tool of diagrams.

6.6. SUMMARY OF RESULTS BY RESEARCH QUESTION

The main objectives of this case study were formulated through research questions, whose answers are summarised below:

RQ1) What practices are involved in using UML in software maintenance projects?

We found a wide variety of practices of UML modelling across different projects in the case company. Projects report that the main purposes of using UML are for 'communication' and 'getting an overview'. The purposes mentioned next most frequently are 'creating a design' and 'own understanding'. This is in line with other surveys (Petre, 2013) about UML and highlights the role of UML as a 'boundary object' –i.e. as a representation of project knowledge used by different stakeholders in different ways. We confirm previous studies which have found that UML modelling is used in a quite loose manner. In our study, we explain some factors that drive the use of UML to a less formal level, which are summarised below:

- Models in project documentation should follow the 'least common denominator'; i.e. they should be understandable to all audiences of the documentation – also those that have no formal training in UML (such as the project manager or stakeholders from business units).
- Software modellers are conservative as regards the use of detailed features of UML syntax, which is prone to change over versions of the UML standard. They prefer to be a little more high-level rather than to need to update the models when new versions of the UML standard appear.
- The main purpose of UML models is communication, and in particular communication of an 'overview of the system' and of 'design intent' – rather than 'design blueprint'. These considerations drive models to focus on key parts of the design. As a consequence, these overview-diagrams intentionally abstract from details. Moreover, in order to convey knowledge/ideas in the best way possible, producers of documentation want to have freedom in which graphical elements to use and freedom to combine freeform text with diagrams. Current UML tools provide very poor support for these combinations.

Overall, because there are multiple stakeholders that use UML models for different purposes, no single perfect UML model exists that is ideal for all. Instead, modelling tools should start catering for different tailorable views for different stakeholders.

With modern software technology, it should be feasible to cater for diverse needs, but the current generation of UML modelling tools does not yet support this.

Some interviewees have asked for executable models or models that support animation. Their claim is that these would be even better for understanding, communication and improving pre-implementation design, but this approach would require a higher level of completeness and detail, and thus incur greater costs. Moreover, it would be more difficult to maintain these models synchronized with the code. In short, the costs-benefits trade-off for executable modelling is unclear.

By and large, reverse engineering is not considered a viable alternative for extracting documentation or models from source code. The benefits claimed for reverse engineering include: i) one does not need to create documentation because one can generate it (i.e. savings of effort), and ii) documentation is always up to date. With regard to argument i): there is ample experience that tells us that reverse engineering cannot be fully automated. Even if some steps can be automated (e.g. identifying all classes and their relations), a lot of manual effort is needed to recover key concepts and abstractions. Furthermore, for some concepts (e.g. sequence diagrams) it is impractical to reconstruct them from the source code because their implementation is scattered across many places in the source code. Moreover, the problems of reverse engineering grow with the size of systems. We therefore found no evidence that savings in effort or development time are made by using reverse engineering models.

Seventy percent of the developers interviewed prefer FD models to RE ones. The situation in which the use of reverse engineering may be practical is that in which programmers need to understand a relatively small piece of the source code that they are working with. In addition to the problem of reverse engineering design models, other concepts are virtually impossible to extract/recover at all from the source code: business processes, design principles, design rationale. Key aspects of such knowledge can thus be documented better.

Finally, there is a negative impact on knowledge sharing: when using reverse engineering to recreate a design model from the implementation (i.e. after the design and programming has been done), this may result in the omission of discussions about the design that would otherwise increase the shared understanding of the design of the systems amongst the developers- i.e. risks of miscommunication is prolonged for a longer period in a project.

RQ2) What are the costs and benefits of using UML in software maintenance projects?

The main cost-factors mentioned for the use of UML actually relate to the change of existing work practices towards using UML, rather than to the actual use of UML. These factors include: cost of training, cost of migrating documentation, and cost of changing processes.

When looking at the actual cost of using UML, the factor mentioned most is that of tooling. With regard to this factor, the quantitative data we elicited at the company

contradicts that tooling is a major cost – e.g. when compared to the costs of training. The costs of tools is therefore perceived to be a high cost factor for large companies, but this is not necessarily so.

It is also interesting to note that there is a radical difference with industries like integrated-circuit (IC) design where design tools can cost up to US $100,000 per year. We think a key difference here is in the purpose: in IC-design the model is a detailed blueprint of the actual IC (hence an engineering construct), whereas in software design, the main purpose of the model is communication. For software modelling, the cost of tooling is perceived as a hurdle, while it is not really so much so in practice.

One factor that complicates the business-case for using modelling is the fact that the benefits are difficult to quantify (they are 'intangible'): no company keeps a record of miscommunication, poor early design choices, out-of-date or unavailable documentation and their associated costs. One of the reasons why such benefits are hard to quantify is that modelling is often one of many ways (employed in conjunction with others) of achieving a particular goal. Faults cannot therefore be traced back to a single cause.

On the benefit side, software engineers from our case study indicate that the use of UML modelling contributes the following benefits:

1. Process benefits:
   - improved communication / fewer misunderstandings – especially across organisational and geographic boundaries (global software development and outsourcing).
   - It helps to improve the design before implementation (through increased ease of peer-reviewing).
   - It prevents knowledge evaporation.
   - It makes diagnosing of problems easier (especially behaviour models).
2. Product quality benefits
   - 89% of the engineers believe that the use of UML improves the quality of the ultimate software product. More specific findings mentioned in support of this are that UML modelling:
   - It increases the understanding of the system to be built.
   - It enables them to monitor whether an implementation conforms to a design.

In general, structural models like class diagrams and component diagrams are not believed to make strong contributions to preventing programming defects in the source code, but they are thought to have a positive impact on the structure of the system (modularity, layering). Modelling therefore makes a better contribution to maintaining a good structure of the system. And this is known to benefit the maintainability of systems. What is more, behavioural models (sequence diagrams) share most of the generic benefits of modelling, but also aid in the diagnosis of errors.

While all of these are mentioned as benefits, there is no empirical evidence regarding the magnitude of their impact on a project for any of them.

RQ3) What are the hurdles when maintaining documentation, and UML models as part of it?

Here we summarise the conclusions obtained as regards both maintaining documentation and using UML as part of the documentation.

Maintaining documentation

A common driver for skipping both creating and updating documentation is time-pressure. Unfortunately projects always have time pressure and there is a lack of evidence to support the benefits of having documentation. Moreover, there is currently a normative challenge in that people (mis)interpret the guidelines of the Agile Manifesto ('working software over comprehensive documentation') in such a way that only working source code is important, and no documentation is needed ('the source code is the documentation').

Other practical hurdles in our case were: the fact that various duplicates of documentation were stored in different archiving systems. This complicated the finding of documentation, as well as the verification of it being up-to-date.

A practical hurdle is that there is often no clear definition of who is responsible for updating documentation. Who should update and when to do so, is often not formally embedded in the development process and there is no quality assurance on updates.

Another hurdle found is related to the misalignment of the incentives for maintaining the documentation:

1. Knowledgeable/experienced developers are required to create documentation, but these are not the people that benefit from it. It is the newcomers and inexperience project members that benefit from the documentation.
2. From a project management perspective, there is also a short-term versus a long-term trade-off. Investing in a good design may lead to easier/cheaper maintenance in the long run. Incentives here are often also misaligned. This could be because the party/part of the organisation that pays for development is often not the same as that which pays for maintenance.
3. The third misalignment is that between those who produce documentation and those who consume documentation. Generally speaking, documentation is created by engineers who have experience with the system they are documenting. This is necessary because they know what the key knowledge about the system that needs to be documented is. But experienced developers do not need the documentation themselves. They produce this for engineers that are relative 'novices' in the system.

Using UML in documentation

One hurdle as regards using UML was mentioned in response to the first research question: it is the hurdle of introducing/migrating to a practice of UML modelling.

Some people/projects that are modelling find it difficult to find a proper level of abstraction and level of detail for using UML models in their documentation. This can be addressed by having company standards and examples that prescribe this. Our

study also shows that different stakeholders have different preferences for viewing – including different LoD – the documentation of a project

Another hurdle is the difficulty of keeping UML and implementation synchronized. UML models in documents are often represented at a medium to high level of abstraction, thus leaving out implementation level information. This choice seems wise from the perspective that this requires few updates to the UML models if minor changes are made to the source code. The downsides are that: i) programmers cannot find detailed guidance in the UML models, and ii) it is unclear which changes to the source code need to be reflected in the UML models.

Given that UML models are primarily used for communication, people not trained in software engineering find the UML notation difficult to understand. Any team will need to find a 'lowest common denominator' for the notation that they use.

RQ4) What are best practices when using diagramming and modelling in documentation?

In our study, the detailed recommendations about best practices can be found in Section 6.6.5. , but we highlight the key practices here:

- PURPOSE OF USE: Find out which stakeholders in a project use UML/documentation and for what purpose. Then have both the parties that produce documentation and those that consume documentation agree on the level of abstraction and level of detail of UML/documentation.
- PROCESS: Clearly define the responsibility for updating/maintaining the (UML) documentation. It should be clear who needs to do this, when this needs to be done, and how. Such practices can be embedded in modern agile approaches –for example, by including updated documentation in the 'definition of done'.
- TOOLING: The archiving of documentation and models should be handled such that it becomes i) easy to find, ii) easy to search within, iii) easy to see if it is up-to-date, and iv) easy to navigate between documents.
- TRAINING: incentivise and/or train in the long term benefits of using modelling languages (especially UML) in the case of those software engineers who are not familiar with these benefits and who do not know any possible benefit of a change towards using modelling in their process. It would also be important to train maintainers on how to use modelling/CASE tools.
- STANDARISATION AND GOVERNANCE: Tooling, training and standardization should be managed at a central level so as to achieve uniform practices across projects and to avoid ineffective attempts at cost savings. Moreover, processes and incentives that ensure that the processes and standards are actually followed need to be instated. Organisations should also create a culture that has a realistic understanding of the value of models and documentation and which thus neither overestimates nor underestimates its value.

6.7. THREATS TO VALIDITY

We must consider certain issues which may threaten the validity of the case study (Runeson et al., 2012). In this section, we therefore discuss the threats to the validity of this study. These threats to validity will be presented in the order of their importance (Wohlin et al., 2000): internal validity, external validity, construct validity, and conclusion validity.

- **Internal validity:** The main threat to the internal validity of this study concerns our ability to control influences from other factors beyond those which have been accounted for in this study. For example, the age of the interviewees, the relationship of the interviewees with their team members, or their motivation, might be influential factors as regards being for, or against, the use of UML. Also the consideration of the term UML as a synonym of Rational Unified Process or even Object Orientation might be an internal threat to validity to this study.
- **External validity**: this concerns limitations as regards generalising the results of a study to a broader industrial practice. The sample of the case study and interviewees might be a threat to the validity of this study, although the sampling process was as randomised as possible. We acknowledge the fact that using only one case study may limit the generalisability of the results of this study. However, we believe that reporting these early findings is necessary, as it serves as an encouragement for other researchers to replicate our study using different case studies. The generalisation of the results might be extended to cases which have common characteristics. On the other hand, an interview provides "spontaneous recall" of an answer if it lists a concrete example/instance on an open question: "*Q: For example, which diagrams do you use?*" "*A: Class diagrams.*" If in a subsequent question, we ask a closed question: "*Q: Do you also use sequence diagrams?*" "*A: sometimes, but not always.*" Then this illustrates that the person actually does also use some other diagram, but needs to be triggered/queued to say so. In interviews it is not always possible to get the interviewee to recall the exact information that is relevant to the question/research. From this perspective, an interview study should not be considered to be complete or accurate in a quantitative sense. Nevertheless, quantitative analyses sometimes provide clear indications of trends.
- **Construct validity**: in the data collection multiple sources of evidence were used. Also the transcript of interviews and observations were sent back to the interviewees to enable correction of raw data. In addition, analyses were presented to them and to the internal research supervisor, in order to maintain their trust in the research. The validity of the developed theory would need to be tested in other case studies.
- **Conclusion validity or reliability**: this relates to the ability to draw a correct conclusion from a study. The chain of evidence from the interviews and documentation analysed through to the synthesized evidence was maintained using a word-for-word transcription. This analysis took a long time to carry out, but this

was due in part to our desire to ensure that we did not make mistakes in the interpretation while the analysis was being undertaken. We therefore asked the interviewees to give feedback to the researchers on the transcripts of the interviews. This practice is known as 'member checking,' and it was used continuously to obtain feedback on both the transcripts and the analyses. Tools were also used during the analysis of the data. Furthermore, the individual coding performed by each researcher was discussed by them, so that they could verify and reach an agreement on them. In particular, we used triangulation in order to reduce bias. Triangulation (Robson, 2002) refers to having multiple sources for the study information. In this study, this was attained in three different ways, which further increases the validity of the study. A summary is provided below:

- Data triangulation: Multiple data sources were used in the study, such as interviews with people who had different roles, experience in ICT, etc.
- Investigator triangulation: Interviews were performed by one researcher, but their analysis was done by two researchers together. Important analysis steps were performed by two researchers independently.
- Methodological triangulation: Multiple methods were use; both qualitative interviews and qualitative archival analysis were employed, along with a few quantitative measures to investigate relevant metrics.

6.8. CONCLUSIONS

In this paper, we have analysed the practices and use of software modelling for software maintenance in industry. We pay particular attention to how UML is being used as part of the documentation available in a maintenance project. This analysis has been carried out by means of a case study involving 31 interviewees playing different roles in a variety of projects in a software department at a multinational company. The qualitative data obtained is presented in conjunction with a theory about how some elements from a common SE approach (like goals, process, practice and tools) influence the documentation or modelling approaches.

The majority of the interviewees consider that diagrams are a helpful tool in software maintenance. The most commonly-mentioned purpose of use of diagramming software designs is 'overview', especially of spatial structures with many components and their mutual relations. Moreover, a graphical notation is an aid to achieving more uniformity of documentation in comparison to textual documentation. It was also noted that diagrams are easy to compare (to other diagrams). Engineers find it easy to judge the relevance of diagrams for their information need, and can therefore quickly glance over diagrams in search of certain pieces of information.

We also detected that the richness of syntactical elements in a notation (like UML) may lead to discussions that do not add any value ('which type of rectangle to use'). This is hypothesised on the basis of discussions about colours in layout that were perceived to be a waste of time.

Moreover, some respondents believe that UML training is a one-time investment (training in UML is needed only once). Other interviewees, however, believe that it is important to have refresher courses in UML, especially for people that use UML occasionally (at a low frequency; several times per year). This would keep them informed about the latest developments in the notation. This instruction could also be triggered by publications of a new version of the UML standard or a significant new release of the tool.

It is probably for this reason that software modellers are conservative in using detailed features of UML syntax that is prone to change in versions of the UML standard. They prefer to be a little more high level rather than needing to update the models when new versions of the UML standard appear. A list of best practices was also extracted from this case study; some of them are based on practices already applied by the interviewees and considered by them as recommendations to people involved in software maintenance projects (or in software modelling), and other are their "wishes". We have classified this list of best practices in five categories: tooling, training, purpose of use, process and standardisation and governance.

Overall, the engineers at the company have a positive attitude towards the use of graphical modelling like UML in software maintenance. Moreover, the discussion should not be framed as 'black-or-white', but as a search for a practical way in which to capture and share knowledge about a system, which will inevitably be a combination of text and diagrams. While quantitative evidence on the pros and cons remains elusive, we believe that the views of the engineers in this company represent a large body of experience.

6.9. FUTURE WORK

Several recent empirical studies into the use of modelling have claimed to be representative of the community of professional software engineers. Our case study highlights that there are many different types of roles (all of them software professionals) in large software development and maintenance projects. However, the involvement of different roles with modelling varies widely. Hence, for future studies, we recommend that researchers pay special attention during the selection of participants in their studies (be they interviews or surveys or experiments), and also in the analysis of their data split, in an attempt to see whether patterns emerge if participants are grouped based on their role. It is not sufficiently detailed to refer to general roles such as 'software engineers' or 'software professionals'.

Possible future directions for our current research are related to the presentation of certain topics which were detected as candidates during the current research, such as:

- Does (just enough) up-front design indeed prevent downstream repair, and to what degree?
- One big question seems undecided: Is the updating of models/documentation time-consuming or not? Or, in other words, how much time do software architects/designers spend on creating and updating models?

- There is a fundamental problem as regards dividing the documentation in a proper way for the purpose of presenting different views to different stakeholders. It might be interesting to research what the proper views for this purpose are.
- Several participants reported that the benefits of creating designs/models are hard to quantify. This area needs new creative approaches in order to attain more insights into this matter.

Our study also identified several problematic areas in current tool support for modelling during software development. There is ongoing research that addresses some of these issues, such as merging and differencing, versioning of models, etc. One issue that stands out is the lack of support for the flexible mixing of diagrams and text. Another issue is the need to abstract models from source code at different levels of abstraction.

Acknowledgements: We are very grateful to the company for dedicating time to us and opening up to us in interviews.

This research has been funded by the SEQUOIA project (Ministerio de Economía y Competitividad), and by the Fondo Europeo de Desarrollo Regional FEDER, TIN2012-37493-C03-01.