Internal Report 2012-08 August 2012

Universiteit Leiden

Opleiding Informatica

Maintaining a software system

with the use of Domain-Specific languages

Tyron Offerman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Introduction

Maintaining software is a huge opportunity to earn money for a company, but it also brings problems with it. Organizations can sign lucrative contracts to maintain software for their clients, because clients cannot maintain the software themselves most of the time. Different departments have to deal with incidents before they can be resolved. Incidents can differ from simple questions to critical bugs. When developing large applications more often traditional programming methods can become quite inefficient when it comes to maintenance. These applications can easily have over ten thousand lines of code in which the structure of the application can easily be lost. Also a lot of code has to be reused, because the same functionality is required in different parts of the application. Maintaining the same functionality in different places can be difficult. Since incidents are often reported on functionality in a certain part of the application, it can be hard to trace back in what other places that functionality also is being used. To give more structure to large applications organizations start looking for alternative developing strategies. One of the methods to turn to, are Domain-Specific languages (DSLs). DSLs can simplify the process of maintaining a software system (1). DSLs can be seen as a method of model-driven development, because DSLs use textual models to develop applications.

This research was conducted at a large IT company that uses DSLs to develop some of their applications. One of the reasons for using a DSL was to improve the maintenance process of their application. In this research a case study is presented where I want to research if the introduction of the DSL did improve the maintainability of an already existing software system. The structure of my research is as follows. First I will take a look at the background of maintaining software with Domain-Specific languages. After the background the case study is presented. Here the organization and how they have implemented a DSL will be told. Also the way the data was collected is explained into detail. This will lead to some results and a conclusion.

Background

Domain-Specific languages can be defined as "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." (2) It can be seen as programming on a higher level on a specific domain. In software engineering a domain can be seen as 'a field of study that defines a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in that field.'[1] For example the financial accounting of an organization is a domain. The terminology is the same for all organizations. If an organization needs software for their financial accounting, the functionality will also be the same. DSLs can be used as a model driven development technique, implementing requirements into the DSL which can be used to generate code. In this case the model is a textual/graphical representation of the system that has to be developed. In traditional programming methods the code often dictates the structure of the application. When an error is reported most of the time a developer or modeler starts looking for the error in the code. Debugging of the system is than done in the code, not in the designs. However with DSLs the model dictates the structure. Debugging should happen within the model. When a problem is reported a modeler should check the designs and should make changes within the model instead of the code.

One of the benefits for a domain-specific language and model driven development is maintainability (3). The system is easier to maintain when underlying technology has to change. Every now and then companies switch to a new technology. When using a DSL for generating an application the designs would not have to change. Only the transformation part of the process has to be changed. The designs have to be transformed to another programming language, instead of reconstructing the entire application, which has to be done with traditional developing methods. This can save a lot of time and effort.

Another benefit is consistency throughout the application (3) something that has been generated will act the same everywhere. For example developing a user interface that has to be uniform and act in the same way throughout the entire application is quite easy to build with a DSL. DSLs can also help keeping business rules consistent. If a business rule has to be used in multiple sections of the application, with some calculations in it, you do not have to develop everything by hand. This way you have less chance of making errors. This will also benefit maintenance because solving a problem in one place, will solve the problem everywhere.

[1] http://domains.askdefine.com/

Case study

This section describes the case study.

Research question

The research question of this thesis is “Does the use of a Domain-Specific language simplify the maintenance of a software system”. This thesis focusses on finding out if the use of a DSL in a real world application did simplify the maintenance of a particular software system. Finding empirical data that supports the theory of a DSL simplifying maintenance (1) is the relevancy of this thesis.

Organization

This research was conducted at a business unit (BU) from a large IT company, with over 5,000 employees and revenues over €520,000,000, in the Netherlands. The organization serves different companies in different domains. This particular BU serves the local government as their clients. The business unit develops an administrative application that supports the daily operations of the local governments in different domains. It’s one standard application that is under constant development by the BU and used by different governments. Governments do not receive a custom made application. However they can modify the system to their needs by selecting the modules they need. Table 1 is an estimate on how many clients the BU has. The first column shows the versions. The second column shows the number of clients with a license. This gives a picture on how many clients the business unit has. Having a license does not mean that a local government also uses that version, because to use a version you need to have the licenses of all the versions before that. The third column shows an estimate on the number of installations, based on the number of clients that have reported an incident. This will give an indication on how many clients have installed the software. However not every client reports an incident.

Table 1: Number of clients that reported an incident

Version	Number of licenses	Number of installations
1	Not available	Not available
2	Not available	143
3	Not available	220
4	Not available	231
5	Not available	246
6	Not available	182
7	Not available	214
8	333	212
9	336	201
10	332	198
11	343	222
12	341	60
13	344	181
14	344	183

In version six the BU introduced a version where a part of the application was developed by using a DSL. In version 12 another development strategy was introduced, which was also based on generating code, to develop the front-end of the application. In this version they started using Microsoft .NET for the front-end instead of Uniface. This was due to the fact that the management of the organization decided that developing with Microsoft .NET should be the leading development strategy throughout the whole organization. The introduction of the new development strategy can also be seen in the number of installations that have reported an incident (Table 1). Version 12 was not as stable as version eleven. Therefore many governments have skipped this release and waited for version 13.

The development team consists of 30 people, of whom 4 are architects (2 architects for the DSL and Uniface and 9 for .NET), 25 are developers/modelers (15 developers/modelers for the DSL and Uniface and 9 for .NET), 1 is a technical writer and 1 is the manager. They are constantly developing the application and have written over 4,350,000 lines of code, this was stated by Bekkers (4) in 2008. These lines of code were developed with Uniface[2] and some C++[3]. 13 years ago the first version of the application had been released. During the life cycle of the application each year there have been one or two releases, with version 16 as the most recent one. Next to the releases there are also patches being released for maintaining the software.

Development process

This paragraph describes the development process that happens before every release, starting with the statement of work and ending with the release.

1) Statement of work
   The number of hours available for the next release is stated in the statement of work. Based on this the BU can decide how many requirements they can complete. Also in the statement of work it is stated which components they want to take into development. Not all the requirements can be resolved within a release, so choices have to be made. Sometimes some components have to be changed, due to a change in the law.
2) National group
   In a national group with representatives of the local governments, the product managers discuss the statement of work. Product managers all have a part of the application or a certain (sub) domain as their responsibility. The group has to decide whether they agree with the statement of work or if they want other components to be taken into development. To make these decisions the group is advised by expert groups. These expert groups have a lot of knowledge about changing laws, which is needed to make a good decision. For example if the way welfare is calculated changes, the software needs to be changed, so it calculates the right amount of welfare somebody deserves. After discussing the statement of work and making some adjustments to it, the requirements are made.

[2] Uniface is a development environment for building, renewing and integrating enterprise applications (8)
[3] C++ is a programming language (8)

3) Basic design
   In the basic design product managers formulate the requirements and how the system should work. Besides internal validation, the expert groups also need to validate if these requirements are correct.
4) Detailed design
   In the detailed design modelers translate the basic design into a more detailed and technical design. This design contains the requirements written in an expression language. This expression language has been designed by the architect that developed the DSL. In a later section I will explain a bit more on the expression language. The detailed design has to be validated internally and externally. After validating the design the development starts. Some parts of the application will be made with the use of the DSL and the other parts of the application will be built with Uniface and some C++.
5) Internal release
   The system, with its new components, will be internally released. Internal testers will test the system and especially the new components. If the test department finds errors within the system, it is send back to the development department. Only if the system passes all the tests it will be released to the pre-pilot.
6) Pre-pilot
   In the pre-pilot some representatives from a few local governments are invited to the office of the business unit. Here the representatives will have a first look on the new release. They can test the system and look if it works the way it supposed to. Small adjustments will be made if not everything is correct.
7) Pilot
   In the pilot the application is installed at some local governments. They will extensively use and test the system. Incidents with the application will be reported. If the incidents are critical they will be resolved before the release.
8) Release
   If all the involved people agree on the pilot the application is released to all the clients.

Maintenance process

Incidents are reported by clients or internally. These incidents will be handled by the support department first. They will try to answer questions and determine if an incident is really a fault or incorrect use of the system. Also the priority of an incident is determined. When an incident is categorized as a fault, it is seen as a change request. The process of incident management is based on Information Technology Infrastructure Library(ITIL), which is standard throughout the IT industry for service management.

At the start of the maintenance process a planning is made by the coordinator of the helpdesk. After the planning has been confirmed by the development department, the patch processes are initiated. This means that all the testing environments are installed and prepared for the patch. The patch process starts. After the patch has been finished it is tested on last time and all the documentation for the release is created, the patch is released to the clients.

During the patch process there are is a process going on at the support department and at the development department. The process at the support department will be described first. The change requests, for maintenance, are checked if they have a proper description and priority. After that the change request is planned as maintenance by the coordinator of the helpdesk and transferred to the incident coordinator of the development department. The coordinator of the helpdesk will check if the change requests keep on progressing. If a change request has been solved, the support department can start testing if it works correctly. If the solution did not pass the test, it is sent back to the development department. If the tester agrees on the solution, the incident report is put in the release notes of the patch.

After the change request is transferred to the coordinator of the development process, he checks whether all the information is in the incident report. If that's the case he will assign the report to a technical analyst, otherwise he will sent it back to the support department. A technical analyst is a modeler with a particular area of attention. They will analyze the report and indicate how the change request can be solved or solve it themselves. If it's a change request that needs to be solved by an architect, the technical analyst will transfer the report to an architect. Otherwise the analysis is given back to the coordinator so he can assign it to developer. The developer needs to solve the change request and document everything. After that he needs to test the solution. If the developer needs the help of an architect, because he or she cannot solve it due to the fact it's a meta-level problem or a meta-level adjustment needs to be made for the solution to work, the modeler can transfer the report to an architect. When the solution has been sent to the support department and the tester agreed on the solution. The developer needs to clear the adjusted components, so others can use them.

If an architect is needed to solve a change request he must analyze if a temporary solution in the meta-level or a work-around can made. If that's possible the architect needs to make the solution otherwise the change request is postponed to a newer version, where the required adjustments in the meta-level can be made.

Some terms that are used in the maintenance process that need clarification:

1) Incident report: The report itself.
2) Incidents: Incidents is a categorization of an incident report. This consists of questions, bugs and wishes. All the incident reports can be seen as an incident.
3) Changes: Changes are a categorization of an incident. Changes are made due to bugs and wishes.
4) Maintenance: Maintenance is a categorization of a change. Changes due to maintenance are all the changes that have to be made to support the current versions that are in production.
5) Developer: Developers are also modelers. However some developers that work with the legacy part of the system often do not serve as modelers. Somebody works as a modelers only if he or she deals with the DSL.

Domain-Specific language

As stated before part of the application has been built with a DSL. Before the introduction of the DSL the application became quite large. Different modules were built completely different from one and another. There was no longer a structure in the software. So there was need for a structured architecture. There was also a need for reusability of the code. Since a lot of functions are used in the same way throughout the whole application it was inefficient to maintain different sources for all these functions. For example the way certain buttons behaved was different defined separately although they behave the same. The BU presented five goals with the introduction of the DSL:

- Speeding up the development process
- Lowering the number of errors in the application
- Increasing the uniformity of the interface
- Increasing the scalability
- Simplifying the maintenance of the application

The goal of this research is to see if the last goal 'Simplifying the maintenance of the application' is reached.

The domain-specific language that the BU uses to develop their application is their own creation. So besides developing the application, they are also developing their own DSL and maintaining it. The first repository of the DSL consisted of 254 files with a total size of 8.50MB (see Table 2). The repository consists of the generic parts that architects develop and the XML files that the modelers create, which will be explained in the next paragraph. These numbers are derived from the version control system. In time the model grew with each version, which can be seen in Table 2. In the latest version the size has already doubled up since the first version. The number of files is close to doubling up. The DSL keeps growing due to increased and improved functionality. It still has room to grow even further, because the estimate is that the DSL generates roughly 30 to 40 percent of the application.

Table 2: Size of the DSL repository

Version	Size of the repository	Number of files in the repository
6	8.50 MB	254
7	9.71 MB	264
8	10.40 MB	293
9	11.32 MB	331
10	12.73 MB	364
11	13.82 MB	383
12	14.73 MB	419
13	15.81 MB	423
14	17.94 MB	442
15	18.31 MB	444
15B	19.17 MB	463
16	19.70 MB	473
16B	19.74 MB	473

From design to source code

Getting from the basic designs to a working application takes a few technical steps. These steps are described below. A graphical representation can be seen in Appendix A: From design to source code.

1) Basic Design
   The basic design is a design made by the product managers in Microsoft Word. A design is a set of tables that are part of one component of the application. In these tables the requirements are written in plain Dutch. The requirements are actually business rules. These are rules that define or constraint specific behavior. (5) For example a client cannot have a date of birth lower than 1/1/1900.
2) Detailed Design
   Modelers create the detailed designs in Microsoft Word. The business rules, which the product managers have put into the basic design, are translated into an expression language, which can be read by a XSLT program to transform the requirements into XML. In the next step I will explain what a XSLT program is and what XML is. The expression language is a language developed by the BU itself to model the business rules. This is the main task of the modelers.
3) XML files
   To create XML that can be transformed using XSLT the word documents need to be transformed to XML files first. XML stands for Extensible Markup Language. It is used for making sure that documents or data structures use a specific set of rules, which you can define yourself. In this case the XML files give structure to the documents. Modelers can do this by using XSLT scripts written by the architects. XSLT stands for Extensible Stylesheet Language Transformations and is used for transforming XML to other data types or transforming other data types to XML.
4) Parsed XML files
   After getting the XML files the files need to be parsed into parsed XML files. This can be done with the use of other XSLT scripts. Modelers simply have to push a button and the files will be parsed. If there are not errors the files are correctly parsed. If there are errors they need to make adjustments in the expression language in step 2 and repeat it again. All the unnecessary data is removed from the XML files and the expression language is transformed into a representation of the data structure. With the help of a XSLT program the word documents will be transformed into simple XML files. The word documents are stripped from all unnecessary data.
5) Implementation/generation
   In this phase the part of the model created by the architects and the part of the model created by modelers are combined to generate the following products:
   a. Functional documentation
   b. Technical documentation
   c. Uniface and Microsoft .NET code
      I. For the .NET code a separate code generator has been built, that creates the front-end of the application.

Role of an architect

The main role of an architect is making sure the DSL progresses through time. He decides what parts of the system are transferred to the DSL. So he's exactly in charge of the policy that the Business Unit follows regarding the implementation of the DSL. Besides that an architect is in charge of keeping all the generic parts of the application up to date. The generic parts of the system consist of the XSLT scripts that are used for transforming the Word documents to XML files, scripts that are used to parse the XML files to parsed XML files and those used for generating the documentation and the code. Another generic part that an architect has to keep up to date is the generic Uniface code.

Data collection

The data was collected through different sources: interviews and the database with all the incident reports. At first I gathered all the available documentation that was written about the DSL. To clarify the documentation, interviews were held with the follow people: head architect, modeler, business unit manager, manager of the software development department and the former manager of the software development. These interviews, which were semi-structured, gave a better picture on how the system has been developed and is being maintained. After understanding the structure of the application and the role of the DSL in it, I knew what to look for in the incident reports. Besides clarifying how the DSL works, it was important to know what the reason was of introducing the DSL and to know what people think of the DSL. Most of the managers did not have a clear picture on what is happening with the DSL. They had a feeling that the goals that were presented with the introduction were reached. They did not have statistics supporting these feelings.

The primary source for the collected data is from the incident reports. Besides clients, these incidents can also be reported by the BU itself. So the data consists of incidents reported by external people and internal people. Due to the fact that version 15 is still in full production, which means that clients are still using this version and therefore maintenance is still being done, I only collect data until version 14. All the incident reports are reported and managed in their own customized system. These can also be exported to comma separated files. Table 3 gives a picture on how many reports there are per category. The dataset consists of over 70.000 incident reports (see Table 3) therefore I have written an algorithm that goes through all the reports and categorizes them in the categories shown in the left diagram of Figure 1. The table also gives a picture of how many changes had been made and how many of them were regarding maintenance.

Table 3: Total number of reports per category

Category	Total reports	Incomplete data
All incidents	76971	3401
All changes	11628	585
Maintenance	4302	24

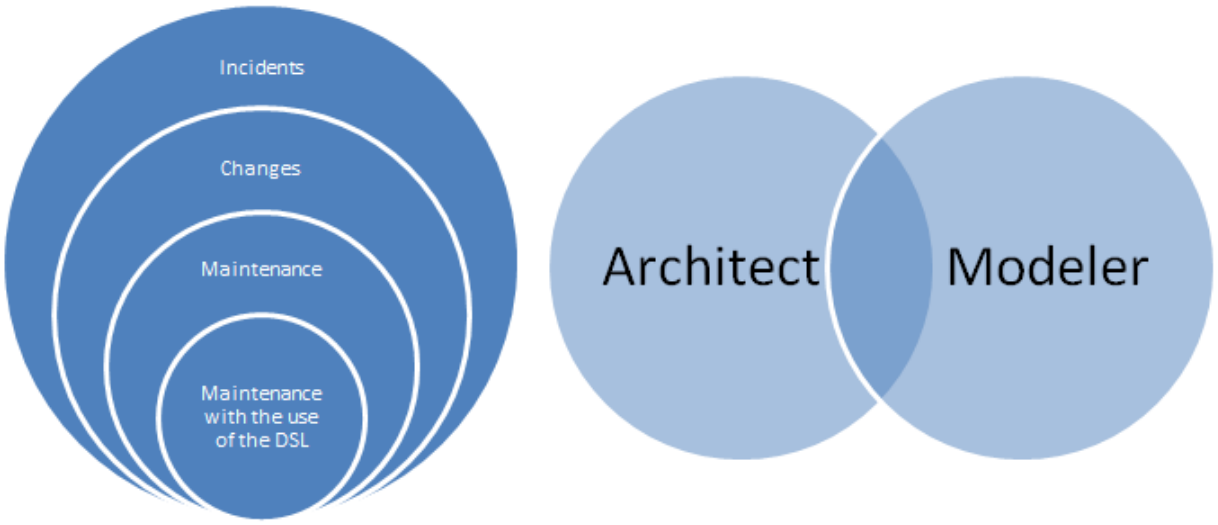

Figure 1: Categories

Every report can be seen as an incident, however not every report as a change. Changes are a subset of incidents. Maintenance is a subset of changes and maintenance done with the DSL is a subset of maintenance. Maintenance done with the DSL means that when a change request has been flagged as maintenance and the way of resolving the issue is by using the DSL. This will help clarify what types of incidents occur at the organization. To answer the research question the latter category (maintenance with the use of the DSL) is split up into three categories, shown in the right diagram of Figure 1. One where architects have to make changes to the meta-level of the DSL, the other where modelers apply changes to the business rules in the model and the intersection of these two categories. This intersection means that a modeler needs an architect to make changes to the meta-model before he can fix the problem or that a modeler needs advice from an architect on how to fix the problem.

To make this algorithm, certain assumptions had to be made to determine how a change due to maintenance has been solved, because there is no uniform way that incident reports have to be filled when solving changes. When an incident was not reported on a specific version or component, it is categorized as an incomplete report. In the section Threats to validity I will go into this further. To see if a modeler made a change in the business rules of the model the algorithm checks whether a design and the parsed version of the design are checked into the database or not. If that combination has not been checked in, a modeler has not made a change within the business rules of the model, so the DSL has not been used. Whether an architect has made changes to the meta-model or not, is determined by checking if the architect has performed any actions regarding the incident report. This information is combined in different ways to derive results. At first it was used to compute results on the number of changes that have been done. Secondly I used the data to retrieve information on the distribution between modelers and architects when the DSL is used. Information about how many incidents had been reported by external people and how many by internal people was computed after that. At last I tried to give a picture on how many versions it takes for a change request to be resolved.

The secondary source of information was interviews. These interviews are different from the ones already mentioned before. Only architects and modelers participated in these interviews. The interviews were semi-structured and being held as an iterative process from the moment the first preliminary results were available till the end of this research. 6 Modelers were interviewed and 1 architect. These were only people that deal with the DSL and not with .NET. The manager of the development department gave the names of the modelers and architect that could be interviewed. It was a diverse group with people from different backgrounds and expertise. Most of the questions were presented as a survey. There were two reasons for doing these interviews: validating if previous mentioned assumptions are correct and finding explanations for the results. Validating if the assumptions were correct and were the best possible practice to gather the right information was quite important. If the assumptions were not correct the results would not have been valid. To reduce bias I did not present my assumptions in the beginning of the interview. This way they had to make the assumptions on their own. In the beginning I assumed that modelers always, not only for maintenance, checked in what files they changed. Although this was confirmed by a couple of interviews, after talking to the incident coordinator, I found out that these assumptions were not true. Modelers only checked in files for maintenance, not for new releases. I also found out that most of the changes that occur in new releases were not reported in the database. Therefore now only information from maintenance was gathered. New interviews revealed that the adjusted assumptions were correct and only minor details had to be adjusted.

To figure out the explanations for the results that I found, the modelers and architects were shown the preliminary results. Questions were asked regarding if they thought the data was correct, that there was no data missing in the results and if they could explain certain peaks in the graphs. Some modelers indicated that the results based on the distribution of the workload between modelers and architects, regarding maintenance were not correct. After some research I found out that the mistake in the data was due to the fact that for architects I use their names to indicate if they had anything to do with an incident, but there is an architect with the same name as a modeler. After changing the search criteria in the database to the last name of the architect, this was resolved. I also asked them if they could relate the results to their own point of view on the DSL and what their personal experience with the DSL was.

Results

In this section the results will be presented.

Terminology

The terminology used in the results is explained below.

- Incidents without changes: Incidents without changes are all the incidents that aren't categorized as changes.
- Changes: Changes are all the incidents that are categorized as changes. The changes can be either product development or maintenance.
- Product development: Changes that are categorized as product development. Product development can also be seen as maintenance with a low priority. This is product development based on incidents. There is more product development going on with the application but that isn't reported in the incident management system.
- Maintenance: Changes that are categorized as maintenance. Maintenance can also be seen as maintenance with a high/critical priority. These change requests need to be resolved as quickly as possible, because the application isn't functioning correctly. It can however happen that a change has a low priority but is still flagged as maintenance. This can happen because the category of an incident can be changed throughout time.
- Maintenance without the DSL: This is maintenance done without the use of the DSL. This means that a change request has been resolved with .NET, Uniface, SQL or C++.The front-end of the system is generated with .NET. Others parts of the system are still developed by writing code in Uniface, SQL or C++.
- Maintenance with the DSL: This is maintenance done with the use of the DSL. Architects or modelers solve these change requests with the DSL. This means that a modeler has changed the input of the DSL, an architect changed the input of a generic part of the DSL or an architect made a change in the meta-level of the DSL. It's also possible that they work together to solve an issue. Sometimes a modeler needs an architect because a generic part or something at the meta-level has to be changed. It can also happen that an architect gets a change request assigned that a modeler can solve then an architect can assign it to a modeler. Another possibility is that a modeler just needs advice from an architect.

The number of changes

Figure 2 shows the distribution of the changes. It can be seen that the percentage of maintenance increases over time to about 80 percent. This is due to the fact that the maintenance flag is used more often than before. Also product development is reported less in the incident report database. Most of the newly added features do not depend on incidents reported and are implemented during the development phase.

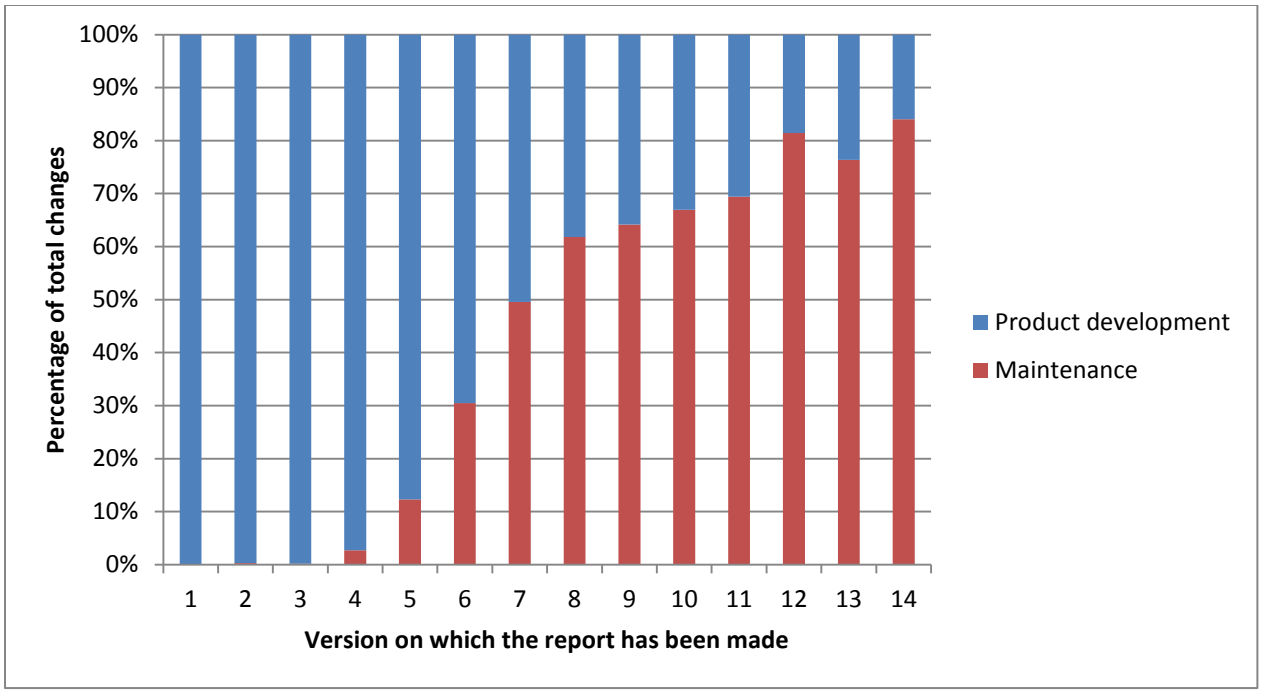

Figure 2: The distribution of changes and maintenance

Looking at Figure 3, which shows the total number of changes that have been made, the total number of changes decreased over time. After the peak in version 4 the number of changes starts decreasing and after the introduction of the DSL in version 6 the number of changes goes down even more and stabilizes around 700 changes per version, until in version 12 the new development strategy was introduced. Because of the new development strategy many governments skipped that version. As stated before many of them waited for version 13. This means that they had version 11 in used for a longer period. Because of the longer period of production, more incidents were reported on version 11 which leads to more changes in that version. Version 12 has fewer changes than version 11, but there were 60% less incident reports in version 12 so there were relatively more changes in version 12, which can be seen in Figure 4. After the release of the twelfth version the number of changes decreases again. The number of changes due to maintenance also starts decreasing again and the balance between incidents and changes is restored.

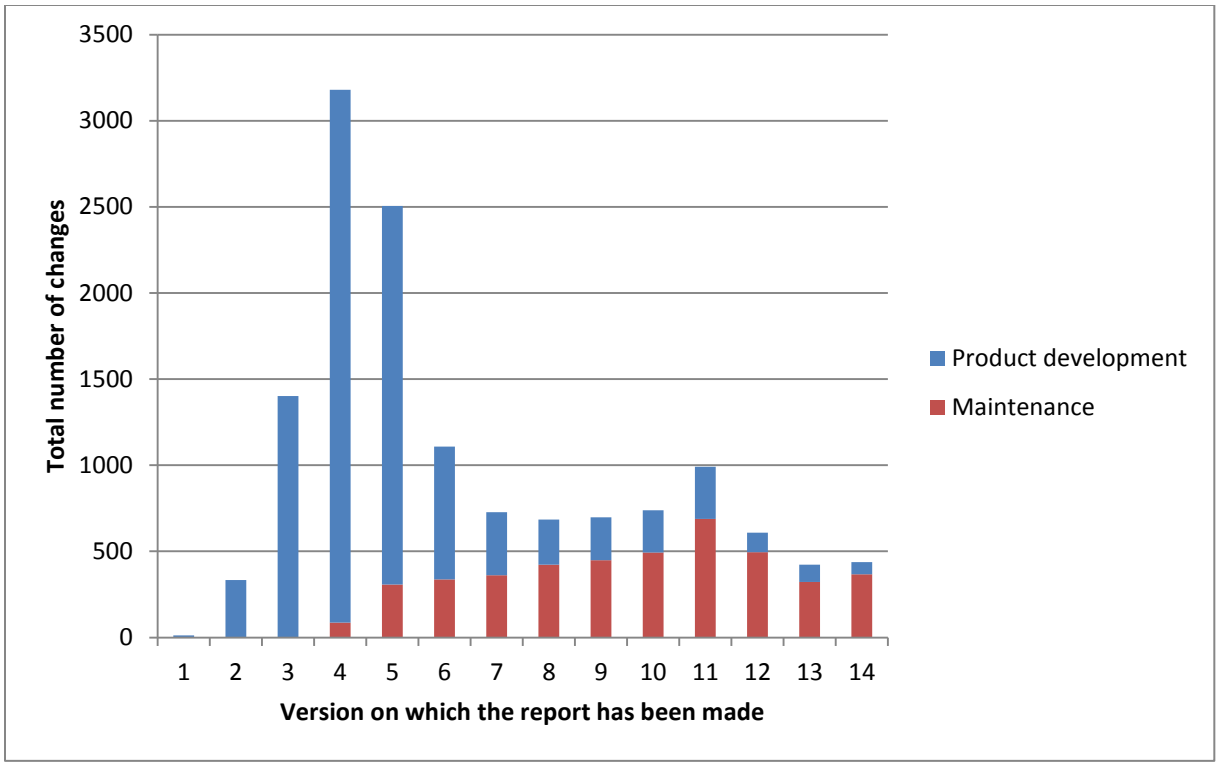

Figure 3: The total number of changes

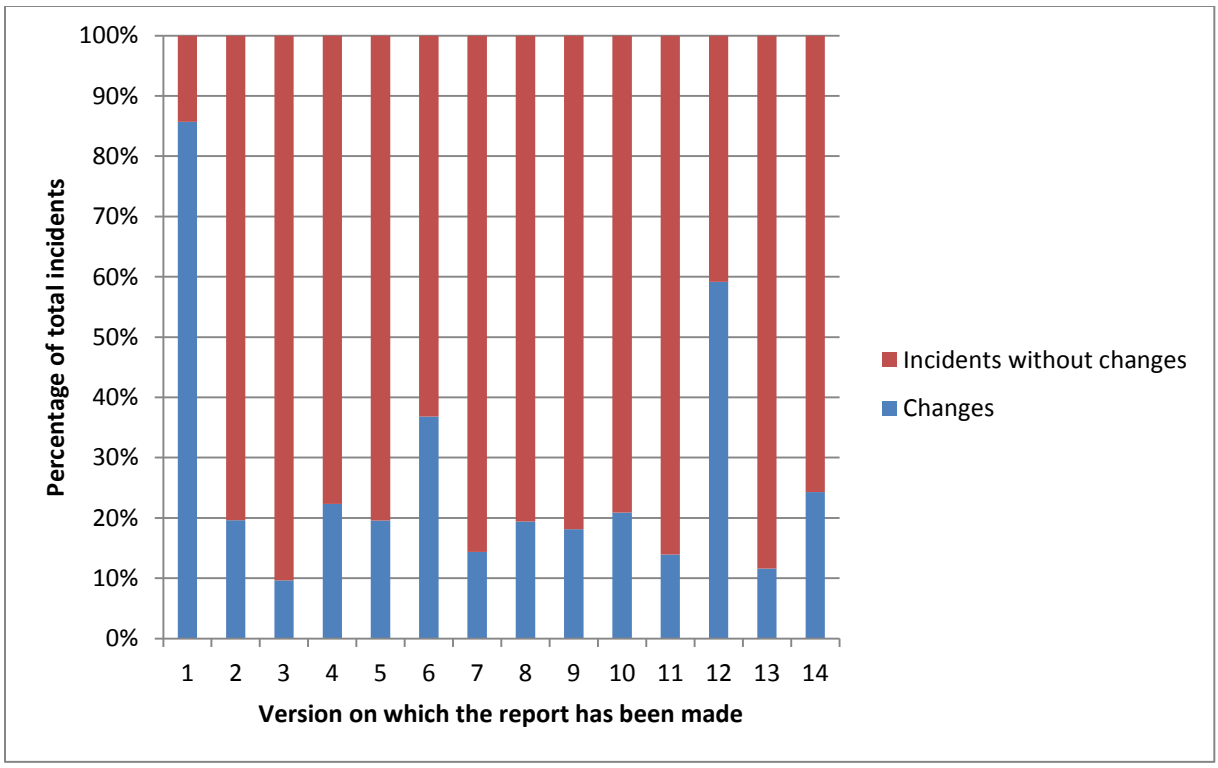

Figure 4: The distribution of incidents

Maintaining the software

Figure 5 shows the distribution and the total numbers of the changes due to maintenance. The percentage of maintenance that is done with the use of the DSL decreases after the introduction of the DSL in version 6 until version 11. From version 11 25% to 30% of maintenance is done with the use of the DSL. This means that in 70% to75% of the time maintenance is done without the use of the DSL. This means the maintenance has been done by traditional programming. There can be different reasons for this fact. One of them is that there are more changes made in the parts that are not part of the DSL. These parts are often quite complex and less error prone than the generated code (2). Another reason is that sometimes the model does not support certain functionality yet. Than a fix is made in the Uniface code for the patch and a real solution will be made in a new version or a modeler can make business rule in the model that says: call my own Uniface code. This last situation can be seen in Table 4 as business rules with services. Most of the time this is part of larger business rule. It's also possible to ignore the model and write Uniface by hand. This can be seen in Table 4 as business rules in Uniface. The data on how the business rules were made could not be retrieved for all the versions, but this will give an indication.

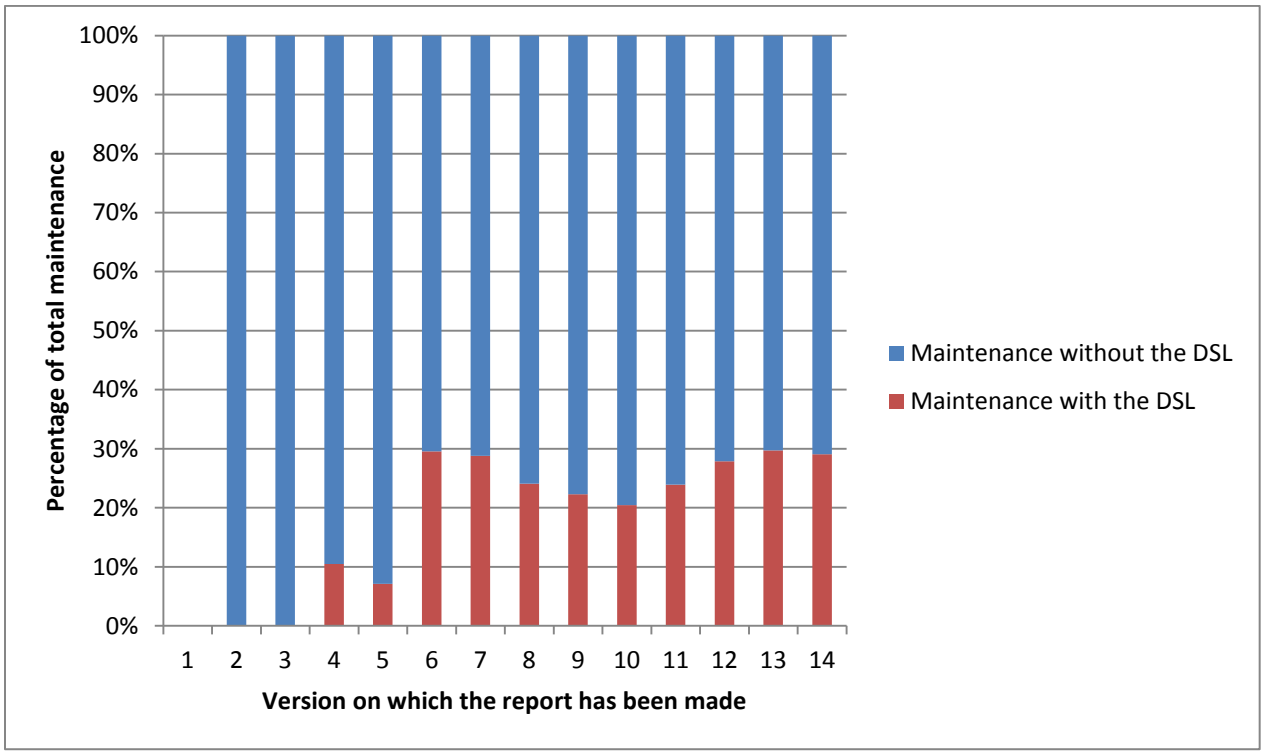

Figure 5: The distribution of changes due to maintenance

Table 4: The number of business rules

Version	# of business rules(BR)	# of BR in Uniface	% of BR in Uniface	# of BR with services	% of BR with services
10	4151	203	4.89	566	13.64
11	4689	217	4.63	603	12.86
12	4864	233	4.79	636	13.08
14	5329	243	4.56	675	12.67
15	5656	250	4.42	701	12.39

Table 4 shows per version the number of business rules, the number of business rules in Uniface, the percentage of business rules in Uniface, the number of business rules with services and the percentage of business rules with services. A business rule in Uniface means that the DSL isn't used for the business rule. These BRs are written in Uniface so these BRs are written manually. This happens most of the time when a business rule is too complex. It's often easier to make complex BRs in Uniface and it can help with the performance as well. A business rules with services means that the DSL has been used for creating the business rule. However a modeler uses Uniface code for a part of the business rule. So these BRs are partially generated.

Figure 6 shows the total number of changes due to maintenance. In this figure it can be seen that the number of changes due to maintenance increases over time, until version 11. After that, it starts decreasing. This was always stated before. The number of changes due to maintenance that were solved by using the DSL is quite stable after the first introduction, with about 100 changes per version. In version 11 there is an increase in the number of changes. However after that it starts declining again until reaches 100 changes per version.

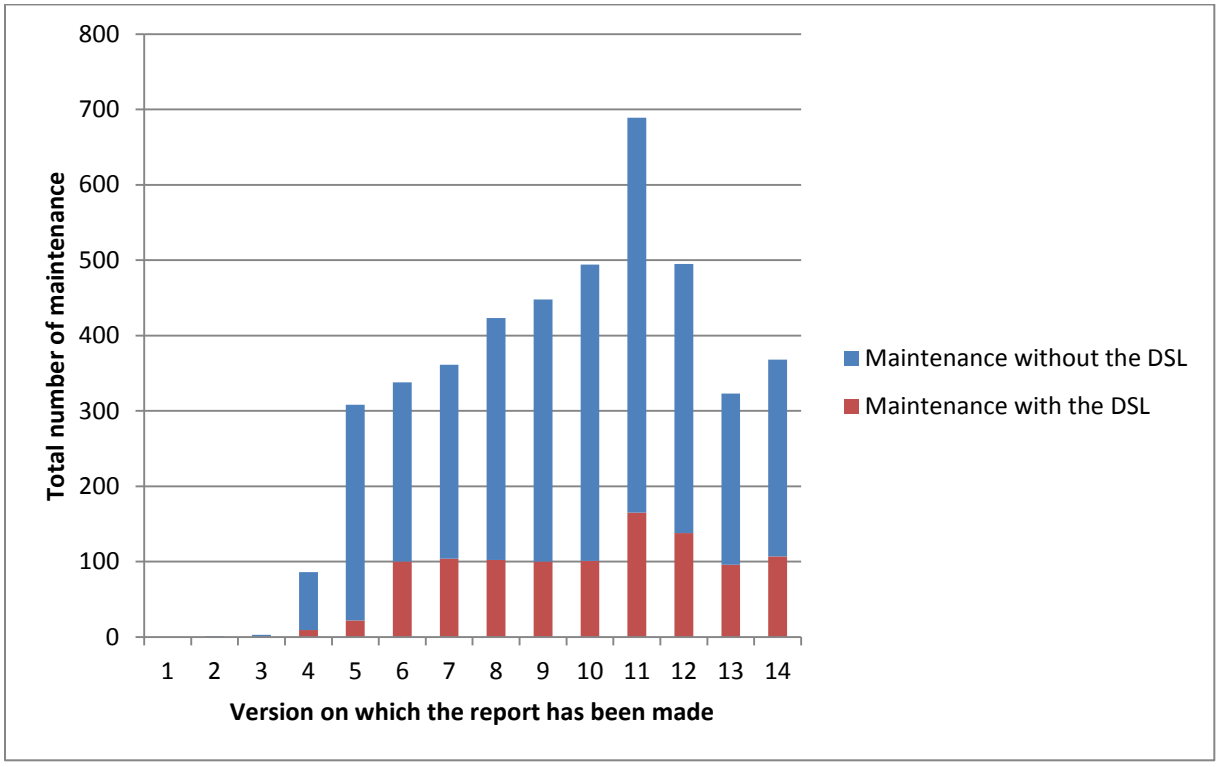

Figure 6: The total number of changes due to maintenance

In the Figure 5 and Figure 6 it looks if the DSL was available in the version before version 6. This is not true however. Due to the fact that not every incident can be resolved within the same version, some incidents are reported on, let's say, version X and resolved in version Y, where version X is not the same as version Y. In section 'In which version are changes solved' I will go further into this.

Distribution between architects and modelers when using the DSL

Figure 7 shows the distribution between the architects and the modelers when the DSL is used to maintain the application. In about 60% of cases where the problems have been resolved by using the DSL a modeler has resolved the problem without the help of an architect. This means a modeler has edited a business rule so that the system will behave in the right way and did not need any (documented) help. Besides the fact that for some things you need the help from an architect to solve a problem, because changes have to be made in the meta-model, it depends on the level of experience on how often you need an architect for advice. The more experience a modeler has, the less he or she needs advice from an architect. This can be seen in Figure 7 due to the fact that the percentage of changes resolved by modelers with the help of an architect decreases since the introduction, with an average of 10%. So modelers get more experienced over time. In version 12 the percentage of changes that the modelers do alone or with the help of an architect goes down. After the introduction of the new development strategy it increases a bit again to the same level as it was before. Some modelers work with the legacy part of the application that has not been implemented into the model yet. These modelers often have less experience with the DSL than the modelers that do most of their work with the model. Through the interviews it came to light that when a modeler is confronted with the model for the first time, he or she might try to solve a problem by working around the model. Sometimes there's no other solution than to work around the model, because some functionality is not supported or the generic functionality is not needed. However when the experience increases and the mindset set is changed to thinking in models instead of code, modelers work less around the model than in the first year since they were confronted with the model.

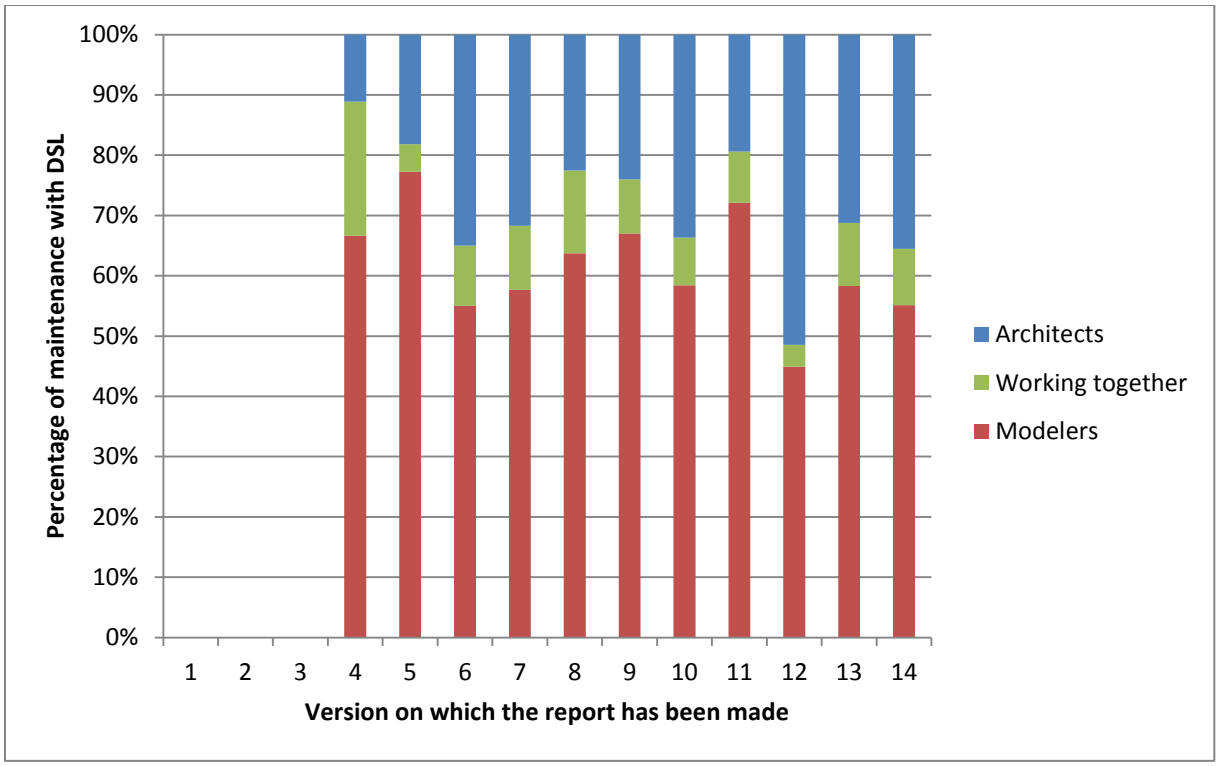

Figure 7: Distribution between changes made by architects and modelers when using the DSL for maintenance

As stated by Heijstek and Chaudron (6) the role of the architect is broader and more demanding, which is shown by Figure 7. This is also illustrated in Figure 8 where in version 12 architects had to make more changes than before, after the introduction of the new strategy. About 40% of the cases an architect is involved in solving a problem. In 20 to 30% of the cases an architect solves the problem on its own. A change in at the meta-level has to be made for the system to behave the way to should, which means an architect has made a change in the meta-model, in the XSLT scripts or in the generic Uniface code. I couldn't compute results for these categories. The data didn't correspond with the reality.

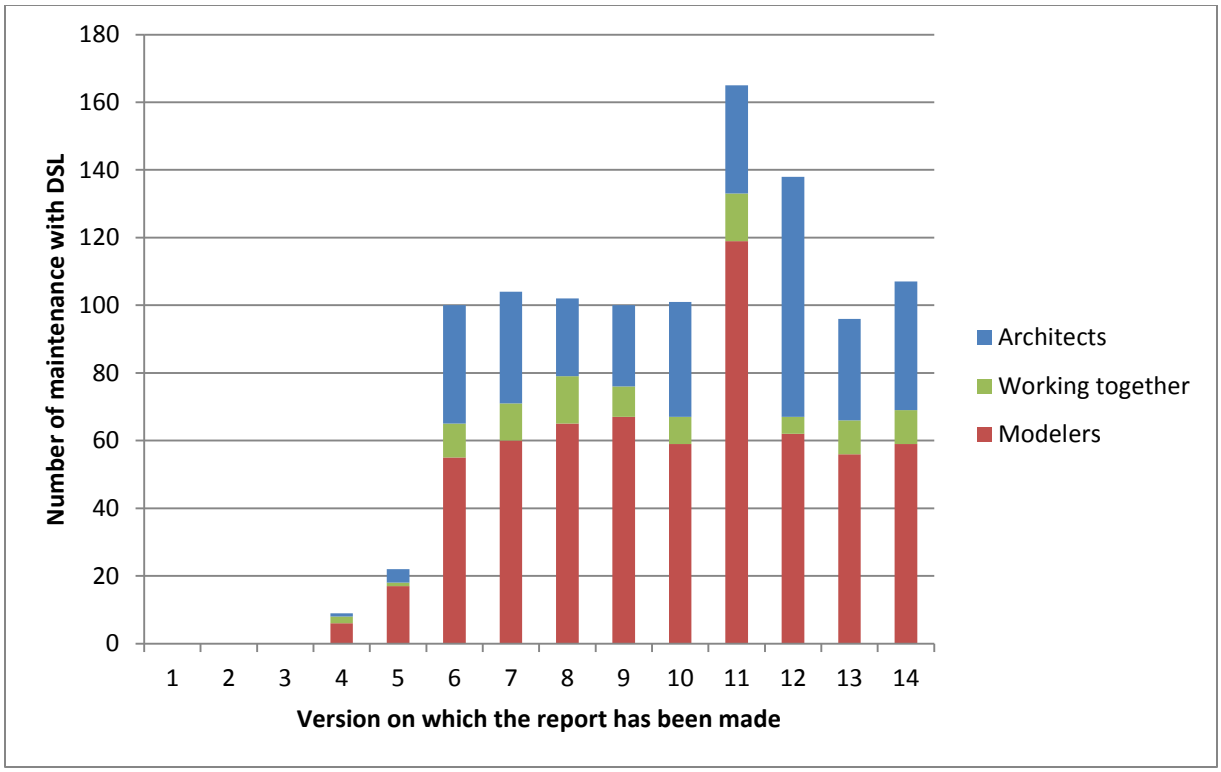

Figure 8: The total number of changes made by architects and modelers when using the DSL for maintenance

Figure 8 shows the total number of changes made by the architects and the modelers. As in the cases before there's a peak in version 11 for the same reasons as mentioned before. The number of changes made by a modeler increases slightly after the introduction of the DSL in version 6. After version 11 it starts decreasing a little bit. Most of the time the number of changes a modeler made is between 55 and 70. The number of changes modelers and architects make together, follows a similar pattern. This pattern can also be seen in the fluctuation of the number of changes, seen in Figure 3.

Internal and external

Figure 9 and Figure 10 show the number of changes that were respectively internally and externally reported. In Figure 9 it can be seen that the total number of changes and the number of changes due to maintenance that are reported internally increases over time, after version 7, until version 13 where it drops. This means that errors are found more often internally than before. The same pattern also exists for maintenance with the use of the DSL. Due to the consistency of the system errors are found more easily. However in these changes there are also pre-pilot and pilot change requests, so the actual number of internal errors found lies a bit lower. These errors are errors that are found after the development department has released a patch or version internally. So the errors found in development are not reported here.

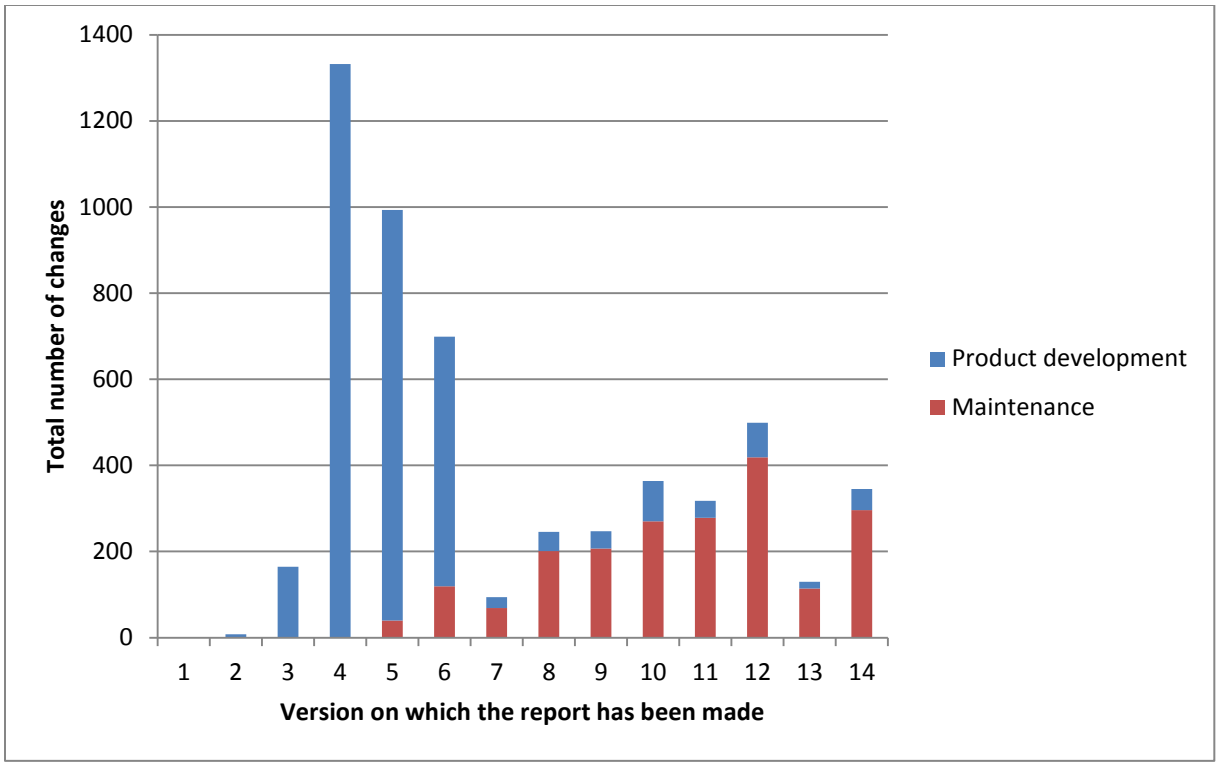

Figure 9: The number of changes reported internally

Even though more errors are found internally over time, the number of changes that were requested from clients is a lot more. This is shown by Figure 10. There is no clear pattern in the total number of changes, except between version 4 and 10. Between those versions the number of changes reported externally decrease over time. The number of changes due to maintenance that is reported externally appears to be quite stable. Looking at both Figure 9 and Figure 10 the difference between internal reported changes and external reported changes is in the number of changes without maintenance. Since the number of changes due to maintenance is almost the same internally and externally.

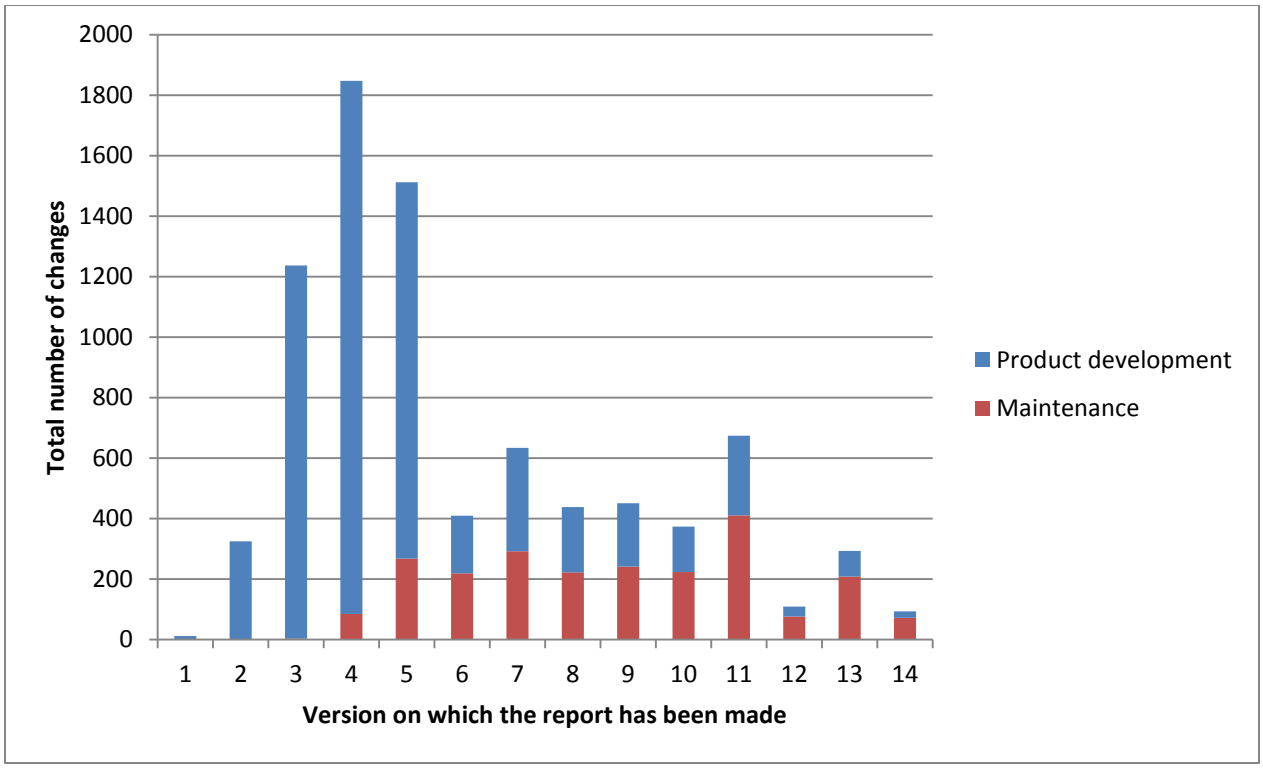

Figure 10: The number of changes reported externally

How long does it take to solve a change

To calculate how long it takes to solve a change two different approaches are used. The first approach was to find out in which version a change has been solved. The second approach was to see how many workdays it takes to solve a change. Both approaches are presented respectively in the next two paragraphs.

In which version are changes solved

If a change request is reported on version X the report can be solved for a patch belonging to that version. However it can also be solved for versions older than version X or for versions newer than version X. Another possibility is that is solved for numerous patches belonging to different versions or to one version. Figure 11 shows these facts, by showing the average version number that a report has been resolved for. For example in version 10 the reports have been solved for one version later. So on average reports that are reported in version 10 are solved for version 11. This means a report can be solved in a patch for version 10, 11 and 12. A change request due to maintenance is often solved for 1.2 to 2.3 patches. If the version in which the report has been solved is higher than the version on which the report has been made, than most of the time there is a newer version already in use, but the client reporting the incident uses a lower version. The issue is than fixed for all the versions in use that are affected. It can also happen other incidents are prioritized and the report is moved towards another version, but is still seen as maintenance and not as a new part of the system. Often patches are also released for a lower version because the lower version is still in use and has the same problems within it.

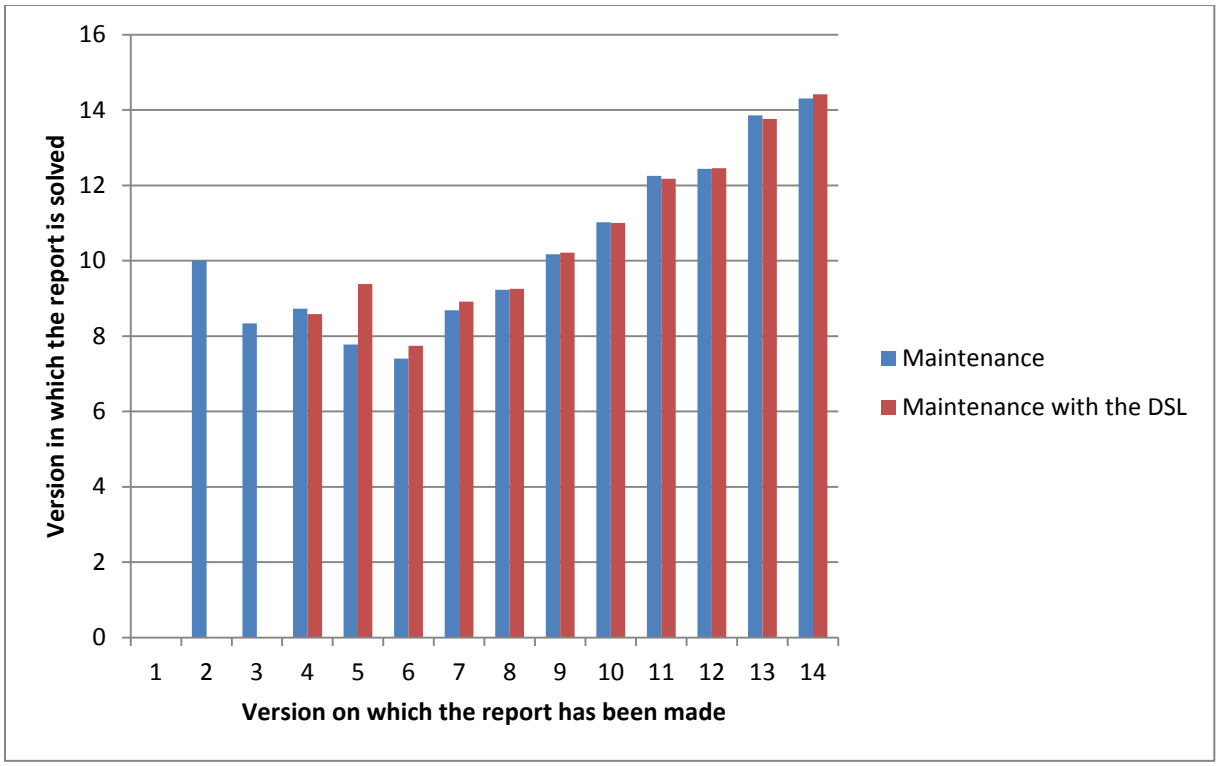

Figure 11: The average version number a report has been solved for per version on which the report has been made

In the early versions it can be seen in Figure 11 that change requests due to maintenance were solved in many versions after that had been reported. This is due to following reason. An incident report in version two did not have a high priority and was postponed to a later moment. The maintenance flag was introduced around version 4 so over versions 2 and 3 there is not more information besides the incidents that were postponed and flagged with maintenance. Since the introduction of the DSL this all stabilizes. However this does not say anything on how fast an incident is solved, but it shows that the total maintenance is solved in the same version number as maintenance solved with the DSL, although the changes made with the DSL are released in more patches, which is shown in Figure 12. Because the part of the application built by the DSL is more consistent a change made with the DSL can be applied to more releases. Also is the chance that an error shows up in more releases than one is bigger.

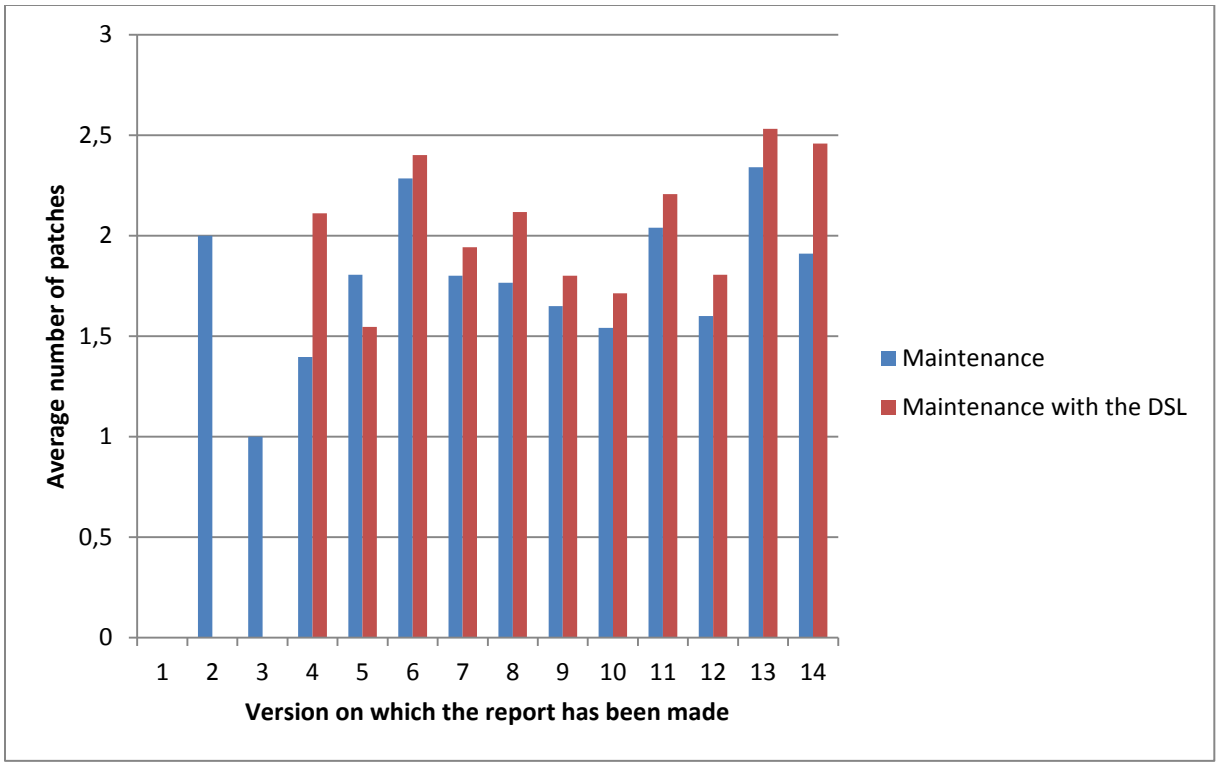

Figure 12: The average number of patches per report

The number of workdays it takes to solve a change

Figure 13 shows the average number of workdays it takes to solve a change request for maintenance. In this figure the average number of workdays for version 2 and 3 are left out. They were so high that the graph couldn't display the other versions correctly. It shows that the number of workdays it takes to complete a change request, from the moment it's reported as an incident till it's solved, decreases over time. The average workdays for total maintenance and for maintenance with the DSL are about the same. This means there's no difference between the two. This also corresponds with the pattern seen in Figure 11.

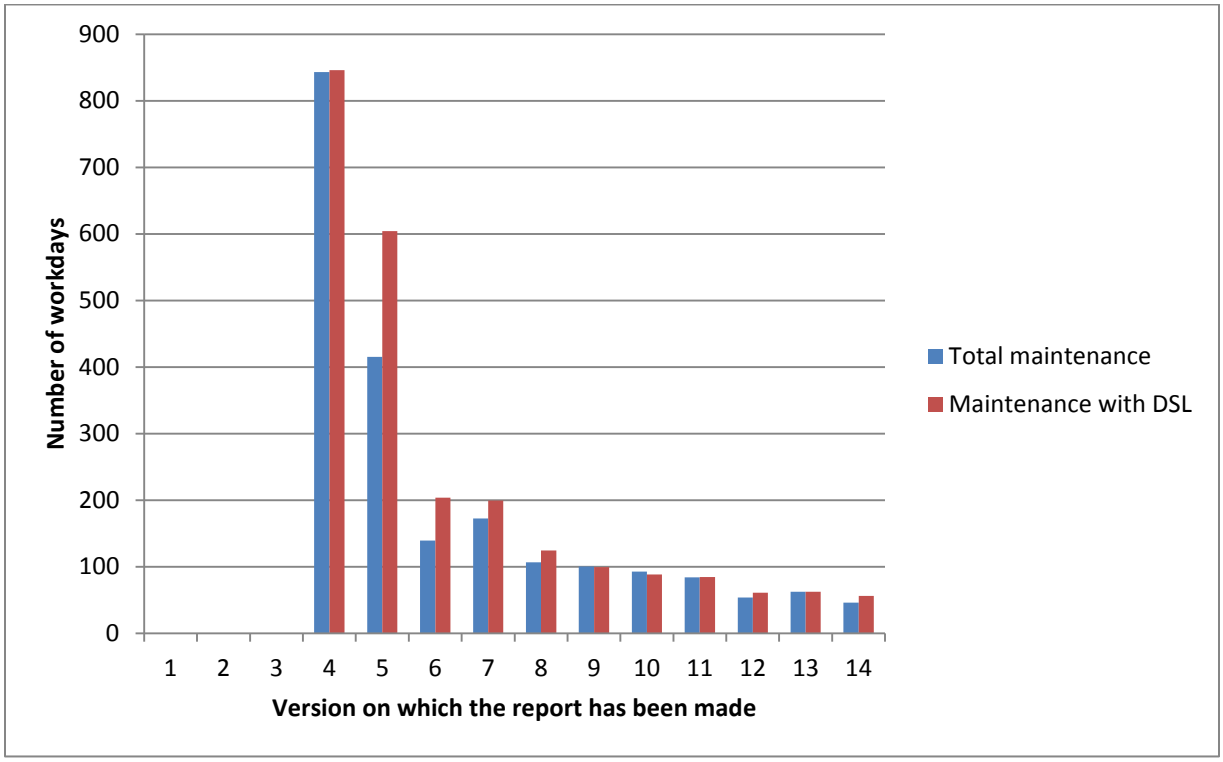

Figure 13: The average number of workdays it takes to solve a change request for maintenance

Figure 14 shows the average number of workdays an architect or a modeler needs to solve a change request. This figure shows that the number of workdays decreases over time. When architects and modelers work individually to solve a problem with the application they take about the same time to solve it. However if a they need to work together to solve an incident it takes quite some extra days to complete these change requests. This is due to the fact that there's an extra layer of work. Another possible reason for this is that modelers and architects don't agree on who should solve the issue. This takes extra time in solving the incident.

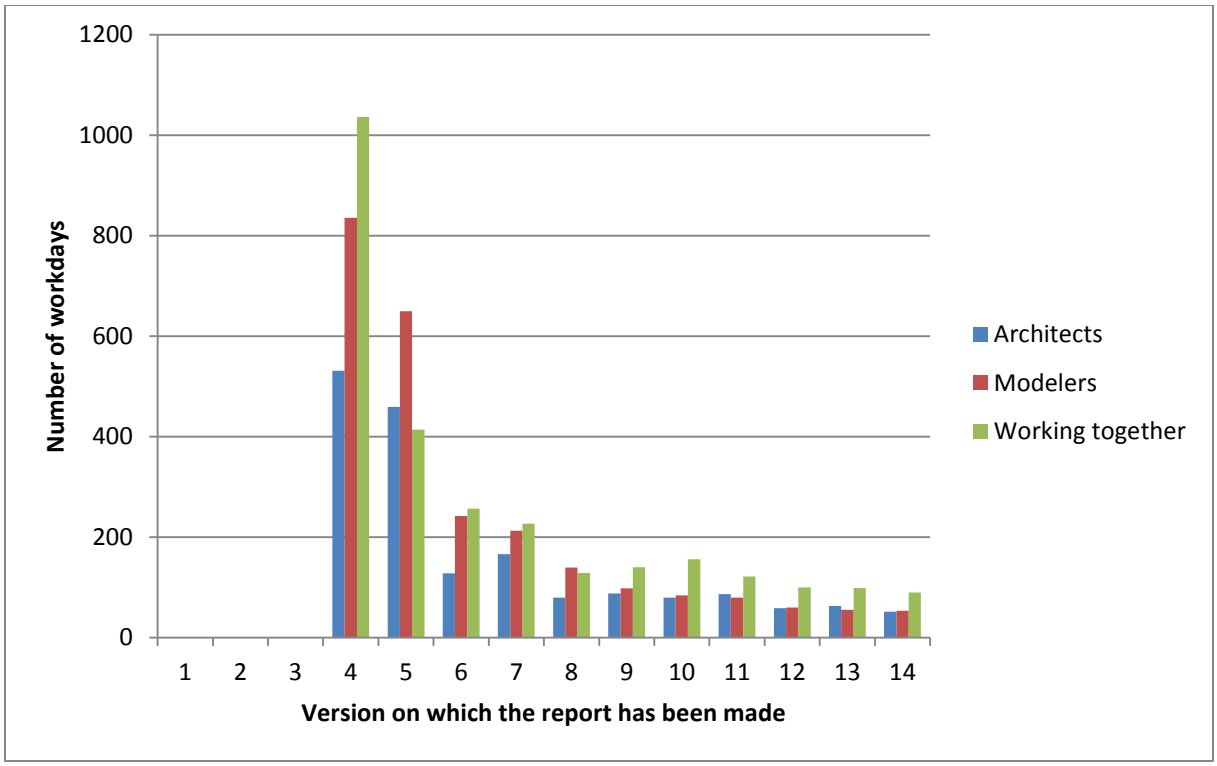

Figure 14: The average number of workdays an architect or modeler needs to solve a change request

Threats to validity

The way incidents are reported is subject to change

Throughout the years the way incidents are reported has changed although the system has not changed. For example the flag of maintenance was not always part of the system. The flag was added between version 3 and version 4. Besides that the meaning of certain words changes. In the incident report system every incident should have a component, although this was not always required, but the meaning of the components can change because the application changes, which might result in components being renamed. It also differs throughout time which elements of an incident report are required and which elements are not. The registration on which version an incident has been reported, can be registered on two ways. One is in the description of the incident, which is quite freely, and the other is a required field where the version has to be filed. As already stated the first way is quite freely, although most of the time the version was registered in the description. The description of the incident was present from the beginning. The more strict way of registering which version an incident was reported on, was not introduced from the beginning. The combination of these two elements should provide a clear picture on the version an incident was reported on.

Data is not always uniformly documented

The way to report an incident is also subject to change, because there are not very strict rules on how to enter data. Especially the way changed documents were checked into the incident report system was not uniformly documented. For new releases almost nothing was documented. For maintenance the changed files are checked in most of the time. Also the information on how an incident is resolved depends on who solved it. Some modelers explain in detail what they did, but others do not do that. This way it might be possible to see who solved the problem, but most of the time it is not possible to see that.

New development strategy

The new development strategy is not part of the results, because incident reports that regard the new development strategy are not well documented. However the impact of the new strategy has been taken into account for the presented results.

Assumptions

Although the assumptions were checked by interviews they are still assumptions. Searching if an architect was involved into solving a problem, does not exactly show what an architect does. If an incident report was assigned to an architect by mistake, the algorithm will count is as an operation by the architect. There was not a way to remove this data since it is not documented who does what exactly.

Single case

This research was conducted on a single case study so deriving general conclusions would not be possible. More implementations of a DSL should be studied to form generalized conclusions on maintaining a software application with the use of a DSL.

Conclusion

In this research the following research question was presented: “Does the use of a Domain-Specific language simplify the maintenance of a software system?” To answer this question a case study was done at a business unit from an IT company in the Netherlands. In this business unit they have an administrative application for the local governments. To develop this application a Domain-Specific language is used to generate the code. In particular the business rules and the user interface are generated. These are rules that define or constraint specific behavior of the application. (5) The results of the case study are based on data from incident reports and are based on interviews held with various people in the organization.

Some benefits of a DSL (3) (1) are confirmed through this case study. The application is more consistent than before. There are no longer errors due to the fact that an interface is not working the same in different parts of the system. For example the way text is inserted will go the same everywhere. The generic code provides the boundaries for this kind of things, making the system more consistent. This helps in maintenance, because these kinds of errors are no longer reported. This has the following effects for maintenance: increased speed of maintenance and lower number of changes. The increased speed of maintenance can be seen as the number of workdays it takes to solve an incident lowers through time. The consistency also helps the fact that a change made with the DSL can often be used in more patches, because the same structure is kept over the versions.

Maintaining with the DSL can be simpler when you only have to change a business rule. Some business rules are more complex, so changing things might even take longer than with traditional developing techniques for those business rules. Those business rules (5% of the total business rules) are therefore made with traditional developing techniques and for most business rules it’s a simple change in these rules to get the right functionality, if you know which business rule to change. This will make maintenance simpler (1). Maintaining with the use of a DSL does require a certain mindset and knowledge of the DSL. However experience grows throughout the years and modelers become more accustomed to the DSL and they can solve almost everything alone.

Also some disadvantages showed up. Modelers need the help of an architect in 10% of the incidents. This means that the role of an architect grows, but modelers are stuck behind. They do not have all the information or the capabilities (yet) that an architect has. This happens more often with modelers that work with the legacy part of the system most of time. These modelers are not really experienced with the DSL and might try to work around the model. If modelers are not thinking in terms of the DSL, but still in code, it can be quite difficult to make changes with a DSL.

Sometimes it’s necessary to make a change in the meta-model before an incident can be resolved. This can be due to the fact that the expected functionality is not supported by the model or that the model generates alternate behavior than expected. There’s exactly an extra layer created when solving some incidents. For example a modeler thinks an architect should solve the incident at the meta-level. After solving the incident at the meta-level the incident is not completely solved, because the modeler has to change some business rules. It also happens that modelers and architects do not agree on who should fix the problem. Incidents are often transferred a lot before solving it. When this

happens, maintaining the system is slowed down when using a DSL. These facts are also the reason that the number of workdays when modelers and architects work together is higher than when they work alone.

So I conclude that the DSL has helped the organization with simplifying their maintenance process in the most cases. Modelers are accustomed to the DSL and use it solve change requests. Also the number of incidents and the number of changes decreased since the introduction of the DSL. The architects have an important role in maintaining the system. When a change in development strategy occurred the architects had to deal with a lot of incidents. These incidents could not be fixed by modelers. Besides that the architects are involved a lot when doing maintenance, not only by changing things themselves, but also as a source of information for modelers. On the other hand the number of externally reported change requests does not seem to decrease over time. This means that customers still find a lot or errors in the application, which should become less over time. The introduction of the DSL also slowed down the maintenance when architects and modelers need work.

Acknowledgement

I would like to express my gratitude towards my supervisor Dr. Michel R.V. Chaudron for supporting my Bachelor Thesis. I'm also grateful that the involved organization and especially the business unit, where I conducted my research, gave me the opportunity to study their use of a domain-specific language.

References

1. *Little Languages: Little Maintenance.* **Arie van Deursen, Paul Klint.** University of Amsterdam : Journal of Software Maintenance: Research and Practice, 1998, Vol. 12.

2. *Domain-Specific Languages:An Annotated Bibliography.* **Arie van Deursen, Paul Klint, Joost Visser.** Amsterdam : ACM SIGPLAN Notices, 2000, Vol. 35.

3. *Model Driven Development – Future or Failure of Software Development.* **Ruben Picek, Vjeran Strahonja.** Varazdin, Croatia : 18th International Conference on Information and Intelligent Systems, 2007.

4. **Bekkers, Willem.** Situational Process Improvement in Software Product Management (Master Thesis). Utrecht : University of Utrecht, 2008.

5. Defining Business Rules ~ What Are They Really? (Chapter 1). *http://www.businessrulesgroup.org/first_paper/br01c1.htm.* [Online] [Cited: 8 22, 2012.]

6. *The Impact of Model Driven Development on the Software Architecture Process.* **Werner Heijstek, Michel Chaudron.** Lille, France : 36th Euromicro Conference on Software Engineering and Advanced Applications, 2010.

7. Rapid Application Development . *http://www.compuware.com/rapid-application-development/.* [Online] [Cited: 8 10, 2012.]

8. C++ Language FAQ - C++ Information. *http://www.cplusplus.com/info/faq/.* [Online] [Cited: 8 10, 2012.]

Appendix A: From design to source code

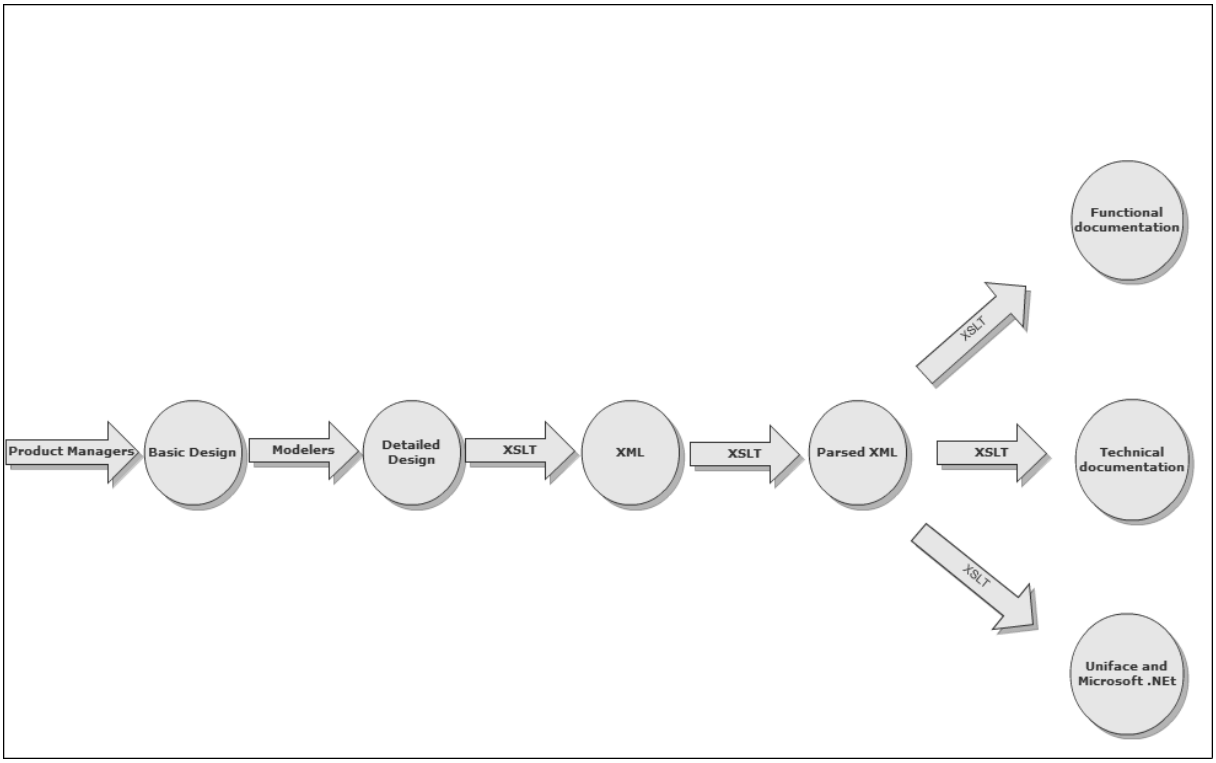

Figure 15: The process of generating code with the DSL