Cost-Effective Industrial Software Rejuvenation Using Domain-Specific Models

Arjan J. Mooij[1], Gernot Eggen[3], Jozef Hooman[1,2], and Hans van Wezep[3]

[1] Embedded Systems Innovation by TNO, Eindhoven, The Netherlands
Email: {arjan.mooij, jozef.hooman}@tno.nl
[2] Radboud University Nijmegen, Nijmegen, The Netherlands
[3] Philips Healthcare, Best, The Netherlands
Email: {gernot.eggen, hans.van.wezep}@philips.com

Abstract. Software maintenance consumes a significant and increasing proportion of industrial software engineering budgets, only to maintain the existing product functionality. This hinders the development of new innovative features with added value to customers. To make software development efforts more effective, legacy software needs to be rejuvenated into a substantial redesign. We show that partially-automated software rejuvenation is becoming feasible and cost-effective in industrial practice. We use domain-specific models that abstract from implementation details, and apply a pragmatic combination of manual and automated techniques. We demonstrate the effectiveness of this approach by the rejuvenation of legacy software of the Interventional X-ray machines developed by Philips Healthcare.

1 Introduction

Software maintenance is crucial to keep up with technology developments such as technology changes (e.g., the shift from single-core to multi-core processors) and technology obsolescence (e.g., the phasing out of the Microsoft Windows XP operating system). Embedded software is often reused in product lines that are developed over a long period of time, but maintaining the existing functionality consumes an increasing proportion of software engineering budgets. This hinders the development of new innovative features with added value to customers.

In industrial practice, it is often considered too costly and risky to make changes to much of the software. As a consequence, software changes are often made by adding workarounds and wrappers to the legacy software. This increases the technical debt [3], such as the size and incidental complexity of the code base, thus making future development even more costly and risky.

Developers usually understand that, sooner or later, legacy software must be rejuvenated to a redesign with a long-term focus; gradual refactoring [10] is not enough. However, a rejuvenation is typically postponed by individual projects due to the time, risks and costs involved. Manual green-field redesign projects often finish too late, require significant additional resources, and are difficult to combine with the short-term innovation needs in highly-dynamic businesses.

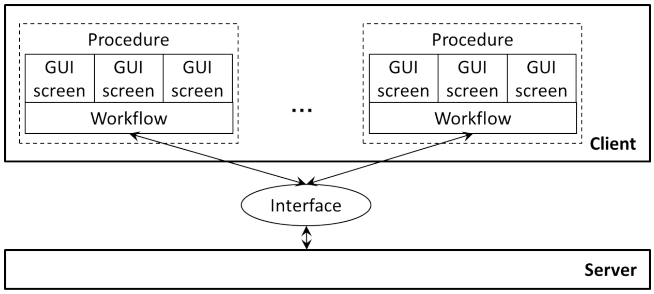


Fig. 1. Software Structure of Field Service Procedures for Interventional X-ray

In this paper we show that partially-automated software rejuvenation is becoming feasible and cost-effective in industrial practice. We demonstrate this using our experiences with the field service procedures of the International X-ray machines developed by Philips Healthcare. The used approach is based on domain-specific models and a pragmatic combination of techniques.

Overview Section 2 introduces the industrial case. The approach is described in Section 3, followed by details about reverse engineering in Section 4 and forward engineering in Section 5. Afterwards, Section 6 treats the industrial verification. Section 7 discusses related work, and Section 8 draws some conclusions.

2 Industrial Rejuvenation Case

Philips Healthcare develops a product line of interventional X-ray machines. Such machines are used for minimally-invasive cardiac, vascular and neurological medical procedures, such as placing a stent via a catheter. Surgeons are guided by real-time images showing the position of the catheter inside the patient.

The calibration and measurement of the X-ray beams is performed by field service engineers. The machines support their work using an integrated collection of interactive field service procedures. These procedures are based on a workflow, in which some steps are automatically performed by the system, and other steps require manual input or action from the service engineers.

2.1 Legacy Software

The software for the field service procedures is structured based on a common separation between user interface and logic; see Fig. 1. For each procedure, the client consists of one workflow and a collection of screens of a graphical user interface (GUI). Each screen consists of a number of GUI elements. The server provides the logic for the automated workflow steps. As depicted at the left-hand side of Fig. 2, the client is implemented using:

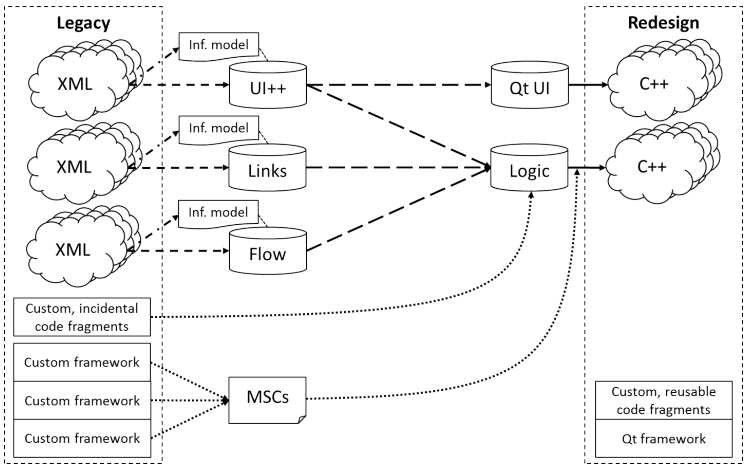


Fig. 2. Rejuvenation Chain for the Client Software from Fig. 1

- three types of XML [2] configuration files (210 kLOC)
- custom, incidental fragments of C# code (10 kLOC)
- stack of custom C++ frameworks (1240 kLOC)

The frameworks are procedure independent and extended over many years. Each specific procedure is configured using the following XML files:

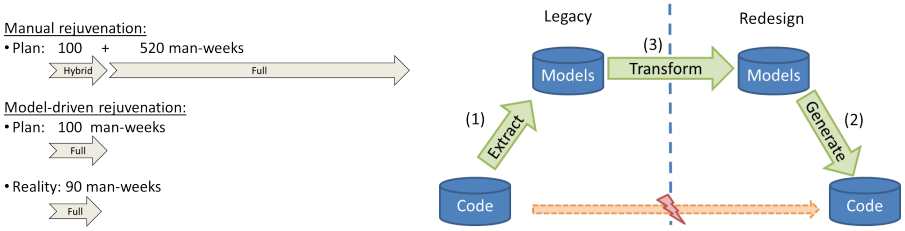
- for each GUI screen, 1 XML file with the static structure (i.e., the placement of GUI elements on the screen) and dynamic behavior (i.e., what should happen if buttons are pressed, items are selected, etc.);
- 1 XML file with links between GUI elements and server identifiers;
- 1 XML file with the workflow and references to associated GUI screens.

The XML files are edited using textual editors, which is time consuming and error prone; see also [9]. The incidental fragments of C# code are procedure specific and are used as workarounds for some framework limitations.

2.2 Rejuvenation Goal

The goal of Philips Healthcare for the rejuvenation of the client software is to make the creation and maintenance of field service procedures more efficient. In particular, there is a wish to eliminate the XML configuration files, and to reduce the large amount of custom framework code (which also needs to be maintained).

The rejuvenated software should be based on well-maintainable C++ code, and off-the-shelf components for common aspects like GUI elements. The main constraint is that the server logic should not be affected by the rejuvenation.



(a) Rejuvenation Effort (b) Model-Driven Rejuvenation

Fig. 3. Rejuvenation of Field Service Procedures

2.3 Business Case

Philips Healthcare continuously develops new innovations for their products. One of these ongoing developments has an impact on the implementation of the field service procedures. However, experienced developers from Philips Healthcare gave high effort estimations for making the required modifications. To make this manageable, the original plan was to divide them over two manual projects; see Fig. 3(a). The first project takes 100 man-weeks and creates a hybrid solution that fits the new technologies with minimal effort. However, this is a typical workaround solution that increases the incidental complexity. The second project takes 520 man-weeks, and should establish the full rejuvenation.

Based on a model-driven approach (see Sect. 3), we have jointly estimated that the full rejuvenation can be completed in the time that was originally planned for the hybrid solution alone (100 man-weeks). As a result the desired rejuvenation could be combined with a regular development project. Thus a strong cost-effective [12] business case has been made for the rejuvenation.

In industry, rejuvenation projects have the reputation of finishing too late. This project took approximately 90 man-weeks over a period of 1 year. It has been carried out by 1 researcher from TNO and 2 developers from Philips, all working 75% of their time on this project.

3 Rejuvenation Approach

The rejuvenation approach that we have used is based on three principles. These range from using multiple information sources and extraction techniques, via the use of domain-specific models, to an incremental way-of-working.

3.1 Combination of Information Sources and Techniques

There are various valuable sources of information about legacy software:

– documentation: human-readable descriptions and diagrams, but usually incomplete and outdated;

– developers: undocumented insights and rationales, but the original developers may not be available;
– code base: very precise, but it is difficult to extract valuable knowledge instead of implementation details;
– running software: the precise external behaviors can be observed, but it is difficult to be complete.

To make reverse engineering effective in industrial practice, we use all sources of information. Some information can more effectively be processed automatically, and others manually; see also [7]. To be cost-effective, we aim for a pragmatic combination of automated and manual techniques.

Reverse engineering of code cannot be fully automated [15]. A particular challenge is to distinguish valuable domain-specific knowledge from implementation details. However, users may have started to rely on certain undocumented implementation decisions in the legacy software.

3.2 Domain-Specific Models

Legacy software was usually developed using traditional methods where the initial development stages focus on informal, natural-language documents. Only in the implementation stage, formal artifacts like code are developed [6]. Currently, software can be developed based on models that focus on the valuable business rules. The implementation details are hidden in code generators.

We build a rejuvenation chain that follows the three steps from Fig. 3(b). (1) The first step is to *extract* (Sect. 4) the valuable business rules from the available information sources, and store them in domain-specific models that abstract from implementation details. (2) The second step is to create a model-driven development environment that can *generate* (Sect. 5.2) the redesigned software from redesigned domain-specific models. (3) The last step in our approach is to *transform* (Sect. 5.3) the extracted models into the generation models.

The used domain-specific models and techniques should be tailored to the specific application domain. The extract and generate steps (including the associated models) can be developed in parallel. The transform step links these models, and hence should be developed afterwards. This order helps to avoid that the rejuvenated software resembles the legacy software too much. Such a model-driven approach is now becoming feasible, because of the improved maturity of tools for model-driven software engineering.

3.3 Incremental Approach

The rejuvenation of legacy software requires a large development effort. In our experience it is important to quickly show successful results to all stakeholders, including higher management, system architects and the developers involved. An incremental way-of-working is pivotal in this respect; see also [18, 14, 8].

In our industrial case, the natural dimension for an incremental approach is the collection of supported procedures. We have addressed procedures in the following order:

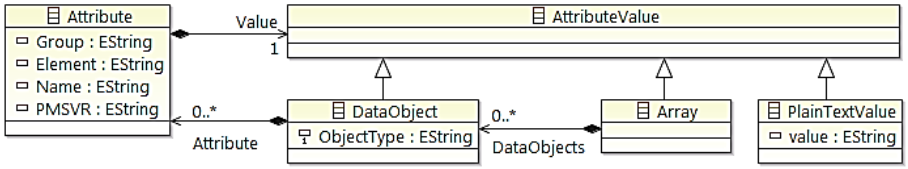


Fig. 4. Schema for all XML files

```
<DataObject ObjectType="PanelProperties">
    <Attribute Group="0x3001" Element="0x652c" PMSVR="IInt16">5</Attribute>
    <Attribute Name="XPosition" Group="0x3001" Element="0x7600" PMSVR="IInt16">0</Attribute>
    <Attribute Name="YPosition" Group="0x3001" Element="0x7601" PMSVR="IInt16">145</Attribute>
    <Attribute Name="Height" Group="0x3001" Element="0x7602" PMSVR="IInt16">35</Attribute>
    <Attribute Name="Width" Group="0x3001" Element="0x7603" PMSVR="IInt16">60</Attribute>
    <Attribute Name="TabOrder" Group="0x3001" Element="0x7609" PMSVR="IInt16">2</Attribute>
    <Attribute Name="LayerType" Group="0x3001" Element="0x77d1" PMSVR="IString">"Data"</Attribute>
</DataObject>
```

Fig. 5. Fragment of an XML File for a GUI

1. some typical procedures (to identify the general patterns);
2. some complex procedures (to identify the variation points);
3. the remaining procedures.

During the first two phases, the complete rejuvenation chain (i.e., steps (1), (2) and (3) of Fig. 3(b)) is developed and extended. In the third phase the rejuvenation chain is quite stable, which leads to an increase in the speed of the rejuvenation project.

4 Reverse Engineering

One of the first steps in software rejuvenation is to reverse engineer the legacy software. The goal is to extract valuable domain-specific knowledge that needs to be preserved, instead of information about how the legacy software works internally.

4.1 Extract Information Model

In the legacy software, the field service procedures are described using many XML files; see Sect. 2.1. The three types of XML files use the same generic XML schema, which is depicted in Fig. 4. A fragment of such a file is shown in Fig. 5. The central element is DataObject, which has an ObjectType and some Attributes. Each Attribute has a data type (called PMSVR) and two identifiers: Name and the combination of Group and Element. The Name is plain text, whereas Group and Element are hexadecimal numbers. The value of each Attribute is described using either a single DataObject, an Array of DataObjects, or a PlainTextValue.

This XML schema is too generic to describe the actual structure of the information. There is a lot of documentation available about the intended information models, based on the ObjectType of DataObjects and the two identifiers of Attributes. This base is still valid, but for the rest the documentation turns out to be outdated.

We have reconstructed the three used information models from the three types of XML files. Using an off-the-shelf XML parser for Java, we have parsed the XML files, and recorded the nesting structure: for each DataObject we record the contained Attributes, and for each Attribute we record the data type and the contained DataObjects. In the terminology of [16], this activity corresponds to "model discovery", although in our case it is not a process model.

The XML files have been edited manually, and hence contain noise. The used XML schema enforces that the information models are duplicated continuously, which naturally leads to inconsistencies. For example, various synonyms were used for the fields ObjectType and PMSVR; the legacy frameworks were robust enough to handle them. Sometimes the two attribute identifiers and the data types were even conflicting. While extracting the information model, we have stored all used patterns, and checked them automatically for inconsistencies. These inconsistencies have been resolved manually in the information model.

Before processing the XML files any further, we have parsed them again and automatically removed the duplicated data about the information model. As a pre-processing step we have also automatically applied the resolutions for the conflicting attribute identifiers, using the combination of Group and Element as the leading identifier. Some other encountered inconsistencies have also been addressed, such as data values of type String where the number of double quotes could be zero, one or two.

The resulting extraction from XML files to pre-processed data is fully automated, but the specific techniques for extracting the information model and addressing the noise and inconsistencies were custom developments. We have developed them incrementally, based on the specific issues that were encountered in the XML files. In Fig. 2, the extracted information models are depicted at the top of the second column. A fragment of the reconstructed information model for the GUI is depicted in Fig. 6.

The extracted information model forms the meta-model of a Domain-Specific Language (DSL [19]) for representing the XML data using Xtext technology. A few fragments of this representation are depicted in Fig. 7(a) and Fig. 7(b). This is more compact, and looks more human-friendly than the original XML.

4.2 Identify Behavioral Patterns

The information model from Sect. 4.1 is used by the stack of custom C++ frameworks to define the runtime behaviour of clients at its two external interfaces: the user interface and the server interface. Given the complexity of these frameworks, we doubt that this behaviour can be extracted (manually or automatically) from the code in a cost-effective way. Therefore we have looked for other options.

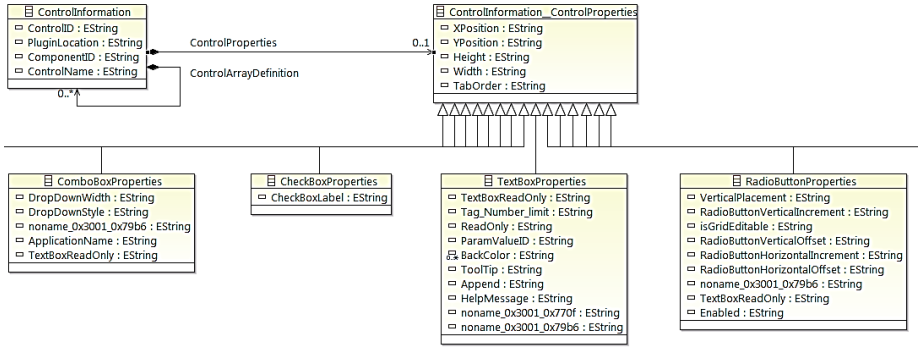


Fig. 6. Fragment of the GUI Information Model

```
ControlProperties = PanelProperties {
    Dock = 5
    XPosition = 0
    YPosition = 145
    Height = 35
    Width = 60
    TabOrder = 2
    LayerType = "Data"
}
```

(a) Part of the extracted GUI (related to Fig. 5 and Fig. 6)

```
NodeTransitionInformation = NodeTransitionInformation {
    InitializationInformation = NodeTransitionInitialization {
        TransitionInfo = Array {
            TransitionInfo {
                ActionName = "NEXT"
                NodeID = "FULLAUTO"
            }
            TransitionInfo {
                ActionName = "Cancel"
                NodeID = "NULL"
            }
        }
    }
    ComponentCreationInformation = ComponentCreationInformation {
        ComponentID = "GENERICTO"
    }
}
```

(b) Part of the extracted workflow

Fig. 7. DSL representation of the pre-processed data

There is documentation available about the interface between the client and the server. This consists of a small, generic state machine and many Message Sequence Charts (MSCs) to illustrate its intended use. The description of the state machine appears to be up-to-date, but it is too generic to accurately describe the behavior of specific procedures. Therefore we have focused on the MSCs.

Most of the MSCs are still valid, so we have used them as base and extended them in various ways. In the terminology of [16], this is called "model enhancement". As the legacy software is still operational, we have inspected the logs of the running software. In addition we have occasionally performed manual inspections of small code fragments. The few observed deviations from the MSCs have been repaired in the MSCs, as the legacy software is considered to be leading.

The result is a compact and very valuable description of the external behavior of the legacy software, combining information from several sources. These enriched MSCs describe how the information model influences the behavior of the legacy software at both external interfaces. In Fig. 2, the MSCs are depicted

at the bottom of the second column. To keep the approach cost-effective, the applied techniques are manual. It is further work to investigate automation using process mining [16] or active automata learning [13].

5 Forward Engineering

The previous section focuses on the legacy software only. The current section transforms the legacy software to a new design using techniques for model-driven software engineering. Before doing so, it first discusses the development of a new software design; thus following the order described in Sect. 3.2 and depicted in Fig. 3(b).

5.1 New Software Design

When developing the new software design, the challenge is to avoid being biased too much by known concepts from the legacy software. Otherwise the rejuvenation may lead to almost a copy of the legacy design. This requires a good knowledge of both modern software designs and the real software requirements (not just the specific implementation decisions of the legacy software).

As described in Sect. 2.2, the redesign aims for plain C++ code and off-the-shelf frameworks. For the GUI we have decided to use the Qt framework, which leads to a design with one class for the static structure of each GUI screen. For each field service procedure, we use one class that contains the dynamic behavior and closely follows the structure of its workflow. To keep the code clean, we have created a small library with reusable code fragments for functionality such as setting up the client-server interface. This structure is sketched at the right-hand side of Fig. 2.

As the field service procedures are based on a notion of workflow, we have considered using a Workflow Management System [17]. We have not done so, because our focus is on a good integration with the existing server, and our goal is to reduce the number of frameworks with a limited contribution. In particular the used workflows are completely sequential and the typical collaboration aspects of workflow management systems, for example, are not relevant in this case.

5.2 Code Generation for the New Software Design

Our aim was to generate the code for the new software design from models; see the third column of Fig. 2. For the static GUI structure, we have used Qt models from which the corresponding code can be generated. The remaining part of the information model describes the logic that is specific for field service procedures: the dynamic behavior of GUI elements, the link between GUI elements and server identifiers, and the workflow. To generate code for that part, we first describe the essential information by means of a DSL. Using modern technology such as Xtext, DSLs can be developed quickly; this confirms observations by [21].

```
// --- node FULLAUTO (stage FullAutomatic) -------
node
    name = "FULLAUTO"
    transitions
        OK -> "STATIC"
        Cancel -> "FINALIZATION"
    stage = FullAutomatic
        part = ""
    UIResource
        assembly = "FSCGeneratorUIDefinitions"
        id = "FSCXGNTubeAdaptationAdjustmentFullAutoUIDef"
    mapping
        event id 21: element "C24"
        event id 20: element "C25"
        event id 10: element "C33" ScrollToLatest
        event id 22: element "C23"
```

Fig. 8. Fragment of a redesigned DSL model

Before developing the DSL and code generators, it is useful to first manually develop a prototype implementation in plain code for a small number of simple cases, without worrying about models and transformations. Based on the prototype code and the required external software behavior (described in the MSCs from Sect. 4.2), the information model for the new design can be identified. Instead of directly reusing the information models from the legacy software, this is an opportunity to eliminate unnecessary complexity. The legacy models are likely to include details about old technologies, workarounds for technology limitations, unnecessary case analysis, and unnecessary inhomogeneity.

Additionally, the new information model might be simplified by making a minor change in the interfaces with adjacent components. In the case of the field service procedures, we have made minor changes in the server from Fig. 1 to solve issues like inhomogeneous data formats. Such changes reduce the complexity of the redesigned client software, without making the server software more complex.

A fragment of such a redesigned DSL model is depicted in Fig. 8. We have aimed for a clean separation between information models with domain-specific knowledge, and code generators with general implementation patterns. This provides a kind of technology independence. After a number of field service procedures had already been migrated, we received the request to change the GUI framework into Qt. The technology independence has enabled us to address this request with very limited effort.

5.3 Model Transformation for the Legacy Software

Finally the legacy models must be transformed into the redesigned models. Parts of these models are closely related, but not all. For example, the legacy models include concepts that deal with internal configuration issues of the custom frameworks; these are ignored in the transformation. Another big difference is the link between GUI elements and server identifiers. The legacy models combine multiple mechanisms for this concept; in some cases there is a direct mapping,

and in other cases it is a combination of two subsequent mappings (described in different XML files). The redesigned models simplify these mechanisms by enforcing the use of direct mappings only.

In a few cases, there was an intricate interplay between the legacy models and the incidental C# fragments. It would have been costly to develop a general transformation for these cases. We have manually provided suitable transformations for these few cases, which turned out to be very effective.

Developing the model transformation is a manual task, but its execution is automated. Instead of a model-to-model transformation, we have used a model-to-text transformation (using Xtext), because the textual representation of the generated models is better readable than their object-oriented structure; see also [8]. By generating text, we can also easily exploit the target DSL's reference resolution mechanism instead of creating explicit references within models.

6 Industrial Confidence

Having finished the rejuvenation, there are three pressing questions. First of all how to verify that the legacy and redesigned software have the same external behavior, secondly the effect on maintainability of the code base, and thirdly how to continue software development from this moment onward.

6.1 Verification

The rejuvenation approach from Sect. 3 does not guarantee correctness by construction. To gain confidence, we exploit an incremental way of working, which enables early feedback. In particular we ensure that all generated artifacts (both models and code) are well-structured; this holds both for final and for intermediate results. In all stages, the generated artifacts can easily be inspected by domain experts to monitor their validity and completeness.

After finishing the rejuvenation, we have assessed the redesigned software using three techniques:

- compare logs of the legacy and redesigned software after performing the same field service procedure;
- review the generated models and code base, which is feasible as they are well-structured;
- execute the legacy set of test cases [12] with good and bad weather scenarios on the redesigned software.

Although the legacy software and the redesigned software have the same functionality, the redesigned software behaves much faster thanks to the removal of several frameworks. This performance improvement has exposed a few bugs (such as race conditions) in other existing software components.

	Legacy	Redesign	
		Model-driven	Plain code
Information models for the GUI	120 kLOC	52 kLOC	—
Generated code for the GUI	—	—	14 kLOC
Information models for the logic	90 kLOC	7 kLOC	—
Generated code for the logic	—	—	27 kLOC
Handwritten code generator for the logic	—	3 kLOC	—
Handwritten incidental code	10 kLOC	—	
Handwritten reusable code	1240 kLOC	5 kLOC	
Total	1460 kLOC	67 kLOC	46 kLOC

Fig. 9. Size of the Software Artifacts

6.2 Maintainability

Maintainability figures are not available yet, but they are often linked to size and complexity. The redesign eliminates incidental complexity from the stack of legacy frameworks; it follows natural domain structures, and the code generation guarantees a consistent style. In Fig. 9 the sizes of the models used in the rejuvenation are summarized, corresponding to the first, third and fourth column of Fig. 2. The sizes in terms of kLOCs are computed using LocMetric; for source code we use SLOC-L, and for the other artifacts we use SLOC-P.

As the Qt and Xtext frameworks are off-the-shelf components, we do not need to maintain them ourselves, and hence they are not mentioned in the table. Concerning the information models, we distinguish GUI models and logic models. In the legacy, the GUI models correspond to one type of XML files that covers both static structure and dynamic behavior; the logic models combine the other two types of XML files. In the redesign, the GUI models are Qt files that cover only the static structure, whereas the logic models are the DSL models.

It is interesting to compare the ratio between various numbers in Fig. 9:

- The XML files for the GUI elements (120 kLOC) have been more than halved to the Qt models (52 kLOC), from which only 14 kLOC code is generated.
- The XML files for the logic of the procedures (90 kLOC) are reduced to 7 kLOC in the Xtext DSL, from which 27 kLOC code is generated.
- The legacy code base (1460 kLOC) is 21 times larger than the model-driven (67 kLOC) and 31 times larger than the plain code redesign (46 kLOC).

6.3 New Software Development Environment

The original rejuvenation aim of the involved managers was to obtain plain maintainable code. However, the rejuvenation chain contains a model-driven development environment for the redesigned software. If further maintenance and development is going to be performed by manually editing the plain code, then it is expected that the situation from Sect. 1 will return.

The developers have decided to continue in a model-driven way. The models that we have introduced can be edited conveniently; for the GUI models there

is a graphical editor (Qt designer), and for the logic there is a textual editor (Xtext). By continuing to generate the code from models, the quality of the code will not degrade.

A potential downside is that this requires the introduction of new tools in the existing development environment. Initially, there were concerns about the learning curve for the developers, but this was no barrier given the modern Xtext technology. Also there were concerns about the possibilities for debugging code. In practice, the software developers had no problem in developing at model level, and debugging at code level, because the generated code was well-structured. The developers have even started to make extensions to the code generators. This confirms the observation [21] that successful practices in model-driven engineering are driven from the ground up.

Thus we provide an interesting way to introduce model-driven software engineering in industry. The model-driven environment is developed as a powerful element in a rejuvenation project. Afterwards, without additional investment, its value can be evaluated for further software development. By storing both the models and the generated code in the software repository, the well-structured generated code always provides an exit-strategy [8] from model-driven software engineering, if this would be desired at any moment in the future.

7 Related Work

The evolution of legacy software to software that uses embedded DSLs is studied in [4]. This work uses the Java-based extensible programming language SugarJ, and gradually replaces larger parts of the legacy software. In particular they present a technique to find source code locations at which a given embedded DSL may be applicable. Our work on software rejuvenation focuses on complete redesigns of a software component (not an evolution but a revolution). Moreover, in our work the DSLs are a means towards a goal, not a goal in itself.

A similar context is used in [7], re-engineering families of legacy applications towards using DSLs. In particular they study whether it is worth to invest in harvesting domain knowledge from the code base. To make our approach cost-effective, we decided that some information should be obtained from the code base, and other information from other sources.

The program transformation work in [1] does not aim to introduce DSLs. They use direct code-to-code transformations, similar to the horizontal arrow at the bottom of Fig. 3(b). This commercial tool has been applied to many industrial cases (including real-time applications), in particular in the context of older programming languages such as Cobol and Fortran. Our work aims for major software redesigns, where domain-specific abstraction steps are essential to eliminate the implementation details.

The model-driven software migration approaches from [11, 5, 20] are based on source code as information source. The source code analysis extracts models that are more abstract than code, based on generic structures like syntax trees. Our work uses multiple information sources and analysis techniques. A crucial

ingredient of our approach is the use of domain-specific models, which abstract from the code and follow the structure of the specific application domain.

8 Conclusion

We have addressed an industrial instance of the software rejuvenation problem. For the field service procedures, the plan using a model-driven approach was 6 times shorter than the plan using a manual approach. The model-driven plan has been realized with 10% less effort than estimated, and combined with a regular project. The rejuvenation has changed the software design substantially, and the software developers have embraced the model-driven infrastructure that was developed to generate the implementation code. The model-driven redesign is 21 times smaller than the legacy code base. Thus we have shown that a cost-effective rejuvenation approach is becoming feasible in industrial practice.

The following recommendations form the key ingredients of the applied model-based rejuvenation approach:

1. work in an incremental way;
2. be pragmatic and aim for partial automation;
3. team up with a large required development project;
4. combine multiple types of information sources;
5. use domain-specific models;
6. eliminate non-essential variation points;
7. generate high-quality well-structured software.

The incremental approach first focuses on the general patterns, then on the variation points, and finally on the bad weather behavior. Such an approach helps to avoid introducing too many (non-essential) variation points from the legacy software. To avoid that the rejuvenated software resembles the legacy software too much, we separate the analysis of the legacy software from the development of the new design. In every increment, the transformation between them is developed as the last step.

This rejuvenation approach is now becoming feasible in industrial practice, because of the improved maturity of the required tools. Domain-specific languages, model transformations and code generators can be developed quickly and easily using modern language workbenches such as Xtext.

We expect that this rejuvenation approach can directly be applied to any legacy software that combines large frameworks with many configuration files, possibly also with some small incidental code fragments. As further work we will consider legacy code in conventional programming languages. This requires other techniques for reverse engineering, but we expect that the same overall rejuvenation approach is applicable. We also plan to investigate to which extent the models with external behavior could be extracted automatically.

Acknowledgment This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project. The authors thank Dirk Jan Swagerman for his trust in this endeavor, and Aron van Beurden and Martien van der Meij for their technical contributions to the software rejuvenation.

References

1. I.D. Baxter, C.W. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of ICSE'04*, pages 625–634. IEEE Computer Society, 2004.
2. T. Bray, J. Paolia, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, World Wide Web Consortium, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.
3. J. de Groot, A. Nugroho, T. Bäck, and J. Visser. What is the value of your software? In *Managing Technical Debt (MTD'12)*, pages 37–44. ACM, 2012.
4. S. Fehrenbach, S. Erdweg, and K. Ostermann. Software evolution to domain-specific languages. In *Proceedings of SLE'13*, volume 8225 of *LNCS*, pages 96–116. Springer, 2013.
5. F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven engineering for software migration in a large industrial context. In *Proceedings of MoDELS'07*, volume 4735 of *LNCS*, pages 482–497. Springer, 2007.
6. J. Hooman, A.J. Mooij, and H. van Wezep. Early fault detection in industry using models at various abstraction levels. In *Proceedings of IFM'12*, volume 7321 of *LNCS*, pages 268–282. Springer, 2012.
7. P. Klint, D. Landman, and J.J. Vinju. Exploring the limits of domain model recovery. In *Proceedings of ICSM'13*, pages 120–129. IEEE, 2013.
8. A.J. Mooij, J. Hooman, and R. Albers. Gaining industrial confidence for the introduction of domain-specific languages. In *Proceedings of IEESD'13*, pages 662–667. IEEE, 2013.
9. T. Parr. Soapbox: Humans should not have to grok XML. IBM developerWorks, IBM, August 2001. http://www.ibm.com/developerworks/library/x-sbxml/.
10. P. Pirkelbauer, D. Dechev, and B. Stroustrup. Source code rejuvenation is not refactoring. In *Proceedings of SOFSEM'10*, volume 5901 of *LNCS*, pages 639–650. Springer, 2010.
11. T. Reus, H. Geers, and A. van Deursen. Harvesting software systems for MDA-based reengineering. In *Proceedings of ECMDA-FA'06*, volume 4066 of *LNCS*, pages 213–225. Springer, 2006.
12. H.M. Sneed. Planning the reengineering of legacy systems. *IEEE Software*, 12(1):24–34, 1995.
13. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Proceedings of SFM'11*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.
14. J.-P. Tolvanen. Domain-specific modeling for full code generation. *Software Tech News (STN)*, 12(4):4–7, 2010.
15. P. Tonella and A. Potrich. *Reverse Engineering of Object-Oriented Code*. Springer, 2005.
16. W.M.P. van der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.
17. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, 2004.
18. M. Voelter. Best practices for DSLs and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.
19. M. Voelter. *DSL Engineering*. Online, 2013. http://dslbook.org/.
20. C. Wagner. *Model-Driven Software Migration: A Methodology*. Springer, 2014.
21. J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, pages 79–85, 2014.