Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation

Arjan J. Mooij, Jeroen Ketema, Steven Klusener
ESI (TNO), Eindhoven, The Netherlands

Mathijs Schuts
Philips, Best, The Netherlands

***Abstract*—Over time, software tends to grow more complex, hampering understandability and further development. To reduce accidental complexity, model-based rejuvenation techniques have been proposed. These techniques combine reverse engineering (extracting models) with forward engineering (generating code). Unfortunately, model extraction can be error-prone, and validation can often only be performed at a late stage by testing the generated code. We intend to mitigate the aforementioned challenges, making model-based rejuvenation more controlled.**

We describe an exploratory case study that aims to rejuvenate an industrial embedded software component implementing a nested state machine. We combine two techniques. First, we develop and apply a series of small, automated, case-specific code refactorings that ensure the code (a) uses well-known programming idioms, and (b) easily maps onto the type of model we intend to extract. Second, we perform model-based rejuvenation focusing on the high-level structure of the code.

The above combination of techniques gives ample opportunity for early validation, in the form of code reviews and testing, as each refactoring is performed directly on the existing code. Moreover, aligning the code with the type of model we intend to extract significantly simplifies the extraction, making the process less error-prone. Hence, we consider code refactoring to be a useful stepping stone towards model-based rejuvenation.

I. Introduction

Embedded software is typically reused in multiple product generations, with changes being made for each generation. Unfortunately, as observed by Lehman [1], "*as an evolving program is continually changed, its complexity ... increases unless work is done to maintain or reduce it*." In other words, software modernization is required. However, this turns out to be challenging. Industrial practitioners [2] indicate hurdles such as limited knowledge of the software to be modernized, which, e.g., makes it hard to identify business logic.

Multiple techniques have been proposed to address the challenges related to software modernization, such as *code refactoring* [3], [4] and *model-based rejuvenation* [5]–[8], with the latter combining reverse engineering (extracting abstract models) with forward engineering (generating new code). In spite of this, challenges remain. Pizka [9] observes that "*the impact of refactoring is limited if the code base has gone astray for a longer period of time*." The risk of introducing bugs is also widely recognized [10], and validating correctness can often only be done at a late stage by testing of the changed or newly generated code [3]–[5], [7], [8].

In this paper, we propose a mitigation strategy for the challenges sketched above. The strategy (see Fig. 1) combines code refactoring with model-based rejuvenation (Extract-Transform-Generate). Specifically, to ensure that the code

- uses well-known programming idioms, and
- easily maps onto the type of model we intend to extract,

we first refactor the code in a number of small, case-specific steps. Once done, we apply model-based rejuvenation.

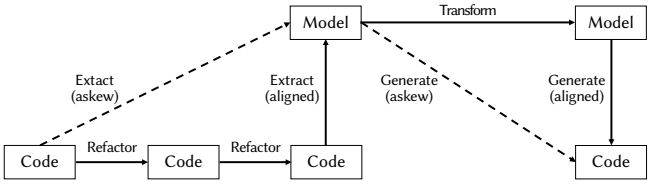

Figure 1. Code refactoring and model-based rejuvenation. The solid arrows represent our proposed strategy, which combines refactoring and rejuvenation. The dashed arrows represent a more traditional rejuvenation approach that lacks explicit refactoring and transformation steps.

Although performing a number of small refactoring steps up front may sound counter-intuitive, as we eventually generate new code, the benefits of this approach are four-fold:

1) The refactoring steps give ample opportunity for *early validation*, as changes can be easily reviewed and tested.
2) Ensuring that the code easily maps onto the type of model we intend to extract *simplifies model extraction*.
3) As low-level code fragments have been tested and follow an idiomatic style after refactoring, we can re-use them as part of the rejuvenated code, allowing us to abstract from these fragments and *focus on the high-level code structure during model extraction*.
4) Aligning the refactored code and the extracted model means that *the extracted model has clearly defined semantics* in terms of the code.

Research Method and Contributions: We conducted the reported research as an exploratory case study that aimed to modernize an industrial software component (see Sect. II). Our main contribution is the formulation of a strategy making model-based rejuvenation more controlled. We report on the strategy in Sects. III and IV in the context of our case study.

II. Industrial Case Study

Our case study concerns a *Controller Area Network (CAN)* adapter from a large embedded system. The adapter, written in C++, was hand coded and implements a nested state machine, with states and substates, that handles incoming messages and

```
 1 StateEnum StateInit::ReceiveResp(Resp* pCanMsg) {
 2   CanModuleStateEnum state = GetState();
 3   m_iReceivedMsg++;
 4   if (m_iReceivedMsg == MODULE_STATUS) {
 5     VersionRequest *pNewCanMsg = NULL;
 6     if (m_bPost) {
 7       m_bPost = !(pCanMsg->GetPostError());
 8     }
 9     if (pCanMsg->GetRuntimeError()) {
10       AddDefectiveControl(pCanMsg->GetControlID());
11       state = FATAL;
12     } else {
13       pNewCanMsg = new VersionRequest;
14       pNewCanMsg->SetID(m_byID);
15       SendMsg(pNewCanMsg);
16     }
17   } else {
18     m_iReceivedMsg--;
19   }
20   return(state);
21 }
```

Figure 2. Coding style of the original code

performs some data processing. The goal is to modernize the adapter to facilitate the implementation of envisioned architectural changes in future product generations.

The state machine consists of 14 states implemented using the state pattern [11], with two states inheriting from a single abstract, partial state. Each state uses up to 13 variables to store state-specific data. The substates, of which there are 69, are distributed over eight states, with the maximum number of substates of a state being 13. The substates are implemented using enumerated types and their C++ integer representation. There are 27 incoming message types.

The adapter has evolved over a long period and functions reliably, but is considered difficult to understand, maintain, and extend. Inspection of the code indicated the use of a number of unconventional programming idioms:

- the integer variables representing substates are often changed multiple times during the handling of a single message and generally hold values that are off-by-one (an increment operation occurs just before every use);
- large non-encapsulated code fragments are used to implement elementary behavior such as sending messages.

Figure 2 illustrates the coding style. The code handles incoming messages of type `Resp` in state `StateInit` and returns the next state. The integer variable `m_iReceivedMsg` represents a substate. Lines 5–16 encode the behavior in substate `MODULE_STATUS` for message type `Resp`.

Intended Model and Extraction Approach: As the adapter implements a state machine, we would like to obtain a state machine model as part of our rejuvenation effort. To this end, we can either apply automata learning techniques or static code analysis techniques. As the adapter performs some amount of data processing, which automata learning techniques do not handle well, we opted for static code analysis techniques.

III. Improving Programming Idioms

As highlighted above, the CAN adapter uses some unconventional programming idioms, hindering understandability. To improve understandability, we began by applying a number of refactorings that substituted the idioms by more conventional ones. The applied refactorings were selected by hand, and were mostly case specific (e.g., because no standard refactorings exist that can turn an off-by-one substate representation into a non-off-by-one representation).

```
1 StateEnum $class::$method($$args) {
2   if (m_iReceivedMsg == $substate) {
3     // REPLACE: $boolVar = $boolVar && $boolExp;
4     if ($boolVar) {
5       $boolVar = $boolExp;
6     }
7   }
8 }
```

Figure 3. An example refactoring specified in our refactoring language

A. Refactoring Technique

After manually selecting appropriate refactorings, we want to apply them automatically, as this allows for repeatability over, e.g., multiple development branches. Moreover, to not interfere with active development, we do not want annotate the source code that is to be refactored to indicate where refactorings should be applied.

1) Refactoring Language: To satisfy the above constraints, we specify our refactorings in a separate language. We do not aim for the language to be fully generic; the language should simply suffice for our industrial case. As the software developers we work with are already familiar with C++, the language is defined as an extension of C++. This allows the developers to review proposed refactorings without having to learn a completely new language.

The refactoring language consists of several elements that express which refactoring operations to apply, to which code fragments in which contexts, and in which order and how often. Figure 3 showcases some of these elements. The refactoring operation is specified via an annotation (line 3) that occurs immediately above the code pattern to which it should be applied (lines 4–6). The context in which the operation may be applied surrounds the operation and pattern (lines 1–2 and 7–8). As a convention [8], an identifier starting with a single `$` denotes a placeholder for a single syntactical element (expression, function argument, ...), and an identifier starting with `$$` denotes a list of placeholders. Although not shown, we use annotations similar to those for refactoring operations to order refactorings and express how often they should be applied.

2) Refactoring Operations: A refactoring operation either specifies how a certain code fragment should be changed, or instructs the refactoring engine to extract certain values for later use. We can also specify side-conditions, e.g., to indicate whether an operation should be applied when the code being refactored has side-effects.

Code-changing refactoring operations come in two varieties:

- replacement of concrete syntax patterns (see Fig. 3), and
- direct manipulation of the abstract syntax tree (AST).

In both cases we build on the C++ parser that is part of the Eclipse C/C++ Development Tools (CDT)[1]. Building on this

[1] http://www.eclipse.org/cdt/

```
1 StateEnum StateInit::ReceiveResp(Resp* pCanMsg) {
2   CanModuleStateEnum state = GetState();
3   if (InSubState(MODULE_STATUS)) {
4     m_bPost = m_bPost && !pCanMsg->GetPostError();
5     if (pCanMsg->GetRuntimeError()) {
6       AddDefectiveControl(pCanMsg->GetControlID());
7       state = FATAL;
8     } else {
9       SendMsg(new VersionRequest(m_byID));
10      ChangeSubState(VERSION_REQUESTED);
11    }
12  }
13  return state;
14 }
```

Figure 4. Coding style after refactoring

parser side-steps the need to write our own, which is a non-trivial task.

Initially, we focused only on replacement of concrete syntax patterns, which software developers find intuitive. However, we observed that some operations are more conveniently expressed at the AST-level, such as:

- inlining methods, to avoid having to specify complete method bodies;
- extracting methods, to avoid having to specify method bodies twice (once for creating methods, and once for identifying locations where calls should be introduced);
- removing unused variables, to avoid having to specify what "unused" means using concrete code patterns;
- combining series of assignments to the same variable, to avoid having to explicitly specify the multitude of possible concrete assignment patterns;
- moving statements backwards and forwards by swapping them with independent predecessor and successor statements, again to avoid the multitude of possible patterns.

B. Industrial Case Study

Although all states and substates of the CAN adapter all have different functionality, their original coding patterns were all very similar, and hence they could be refactored in similar ways. We applied the following refactoring steps in order, where steps 3–5 depend on steps 1–2:

1) removing dead code, and declaring local variables as late as possible in the closest encompassing scope;
2) ensuring that updates to (sub)states occur as late as possible and are not mixed with the sending of messages;
3) replacing updates to substates via integer operations with direct assignments of the members of the relevant enumerated types, and ensuring these are not off-by-one;
4) encapsulating all operations required for message creation in message class constructors;
5) making logging homogeneous, and introducing auxiliary methods encapsulating data processing.

The result of applying the refactorings to the code of Fig. 2 can be found in Fig. 4. The first refactoring is obviously generic, while the others are case specific. All refactorings aim to improve the understandability of the code.

The individual refactorings were easily validated (by means of code reviews and existing test suites), and integrating the results into the code base went smoothly. The refactorings could also easily be adapted to other development branches, and the developers of the adapter considered the refactored code to be much easier to understand.

C. What We Learned

We learned the following while refactoring the adapter:

- specifying case-specific refactorings is both about the operations and the context in which they are applied;
- specifying refactorings is sometimes done best at the concrete syntax-level, and sometimes at the AST-level;
- case-specific refactorings enable code changes that are only valid for the specific case;
- incrementally applying small, case-specific refactorings facilitates early validation.

IV. Redesigning the High-Level Structure

The refactoring steps from the previous section addressed the main issues with the CAN adapter identified in Sect. II. However, the steps did not improve insight into the overall behavior of the implemented state machine. To improve insight, we next visualized the state machine by extracting its high-level structure (again using CDT's C++ parser), while ignoring low-level details. The visualization provided some new insights and could be kept up-to-date by regenerating it, but its size was substantial given the number of states and substates, which still made it difficult to fully comprehend the state machine.

We next investigated whether we could improve the high-level structure of the code to reduce the need for a separate visualization. The key observation we made was that the code was structured from the top down (i.e., viewed from class definitions, via method definitions, to control statements) as

state → *message type* → *substate* → logic

whereas the visualization was structured as

state → *substate* → *message type* → logic.

Although the top-down structure of the code reduces code duplication when incoming messages are handled similarly across multiple substates, the top-down structure of the visualization is more useful when trying to understand the code's behavior in terms of the order of operations from entering a certain substate, via processing of various incoming messages, to exiting the substate.

To transform the top-down code structure to match the one used in the visualization, we first created a model by extracting the high-level structure of the code. Thereafter, we transformed the model to give it the appropriate top-down structure, and we generated new code by combining the transformed model with low-level code fragments that we had retained.

A. Rejuvenation Technique

Building again on CDT's C++ parser, we parse the relevant source files without expanding any `#include` directives. We then extract a high-level model that is close to the structure of the original source code, using pattern matching on both concrete syntax patterns and AST patterns. We retain low-level

code details by saving relevant code fragments along with the high-level model, as proposed by [8].

To validate the extraction and give the extracted model semantics, we develop a code generator in Xtend[2] that is able to regenerate the original code (apart from formatting) from the high-level model and the retained low-level fragments. Thereafter, we transform the model to obtain the desired high-level structure, and use Xtend to generate new code from the transformed model and the retained low-level fragments. Lastly, we use CDT's code formatting capabilities to format the new code (to nicely integrate the retained fragments).

B. Industrial Case Study

Once we started to extract a state machine model from the CAN adapter code, we realized that although we did improve the structure of the code while refactoring, extraction was not as straightforward as it could be. Therefore, before proceeding, we applied a number of additional refactorings:

- introducing explicit state assignments in simple cases where the state does not change (for homogeneity);
- replacing if-statements with guards containing a disjunct with an `InSubState` call by a chain of if-then-else statements (to isolate `InSubState`);
- replacing if-statements with guards containing a conjunct with an `InSubState` call by nested if-statements (again to isolate `InSubState`);
- moving calls to the `InSubState` method out of helper functions, by inlining fragments of these functions.

The above refactorings were geared towards making the state machine, and hence the high-level structure, more explicit. We did not perform these steps earlier, as they somewhat increased the number of lines of code.

Once we applied the above refactorings, model extraction was straightforward, with the inheritance from partial states being the only source of complexity. After extraction we transformed the model to match the top-down structure of our visualization, and generated new code.

Figure 5 presents the result of applying the above steps to the code of Fig. 4. The depicted method handles all messages that can be received in substate `MODULE_STATUS` of state `StateInit`. Of course, as the steps changed the high-level structure, their impact is difficult to gauge from code fragments only. However, the adapter's developers considered the code much better to understand even without a separate visualization, although code size increased slightly.

C. What We Learned

We learned the following while rejuvenating the adapter:

- aligning the original code with the type of model we intend to extract simplifies the extraction, and also enables early validation through easy regeneration of the code;
- retaining low-level code fragments and linking these to the extracted model allows us to focus on the high-level behavior in the model;

[2]https://www.eclipse.org/xtend/

```
1 StateEnum StateInit::ModuleStatus(Msg* pMsg) {
2   CanModuleStateEnum state = GetState();
3   if (dynamic_cast<Resp*>(pMsg) != NULL) {
4     Resp* pCanMsg = dynamic_cast<Resp*>(pMsg);
5     m_bPost = m_bPost && !pCanMsg->GetPostError();
6     if (pCanMsg->GetRuntimeError()) {
7       AddDefectiveControl(pCanMsg->GetControlID());
8       state = FATAL;
9     } else {
10      SendMsg(new VersionRequest(m_byImageID));
11      ChangeSubState(VERSION_REQUESTED);
12    }
13  }
14  return state;
15 }
```

Figure 5. Coding style after refactoring and rejuvenation

- code refactoring can act as a useful stepping stone towards model-based rejuvenation;
- changing the high-level code structure can reduce the need for separate visualizations.

V. Threats to Validity

Threats to *internal* validity come from the way in which we carried out our case study. Our study focused on qualitative aspects and ignored quantitative aspects. The time to create the case-specific refactoring steps was not considered.

Threats to *construct* validity come from the way in which we evaluated our case study. We did not attempt to transform the considered adapter using different approaches.

Threats to *external* validity come from the way in which our results will generalize to other software components. Our case study only considered a single adapter. We believe that the presented approach could have helped in case studies such as [8], but more research is needed to determine its generality.

VI. Related Work

We discuss various avenues of related work.

A. Model-Based Methods for Software Modernization

The literature review from [12] compares in detail fifteen different model-driven reverse engineering approaches, and observes that the approaches and applications are diverse.

Industrial case studies on model-based rejuvenation are presented in [5], [8]. The study from [5] employs models based on generic concepts such as data structures, algorithms, and GUI elements. Any source element that does not easily fit the model is reported to the user. The study from [8] employs domain-specific models, and inspired us to retain low-level code fragments during rejuvenation. Contrary to us, the authors of [8] do not first attempt to align the code with the type of model they intend to extract.

An industrial case that combines model extraction and code refactoring is presented in [13]. Architectural models are extracted to give insight into code structure, after which desired model changes are specified. Next, case-specific code refactorings are derived, but no rejuvenation is performed.

B. Code Refactoring Tools

Most refactorings are performed in batches [14], and developer-specified refactorings are sometimes seen as the Holy Grail of refactoring [15]. Unfortunately, tooling for developer-specified refactorings is currently lacking [10].

Frameworks for specifying refactorings generally only allow refactorings to be specified at the AST-level. Examples of such frameworks are Clang's C++ refactoring engine[3], the Java refactoring engine presented in [16], and the MoDisco framework [17], which has been used to perform large-scale Java refactorings.

Specifying C/C++ refactorings using concrete syntax patterns is supported by a limited number of tools. Coccinelle [18] enables developers to define C refactorings, and Rascal [19] supports C/C++ via its ClaiR[4] module [20]. DMS [21] and TXL [22], which focus on program transformations including restructuring, also support concrete C/C++ syntax patterns, and have been used in commercial applications.

C. Extracting State Machine Models from Code

To enhance insight in software, [23] describes a method for extracting visual representations of state machines implemented in C. The method is based on matching specific implementation patterns. Nested state machines and conditional state transitions are left as future work, although the authors do give examples that exhibit a similar kind of top-down structure as the state machine from our case study (see Sect. IV).

An industrial case study on extracting flat state machine models from C code is described in [24]. The approach taken in the study is similar to ours, namely, recognizing occurrences of specific implementation patterns that are used consistently in the considered code base. Semi-automated techniques for extracting state machines from C code that does not follow specific implementation patterns are presented in [25] in the context of embedded control software.

VII. Conclusions and Future Work

As we have shown, code refactoring can be used to make model-based rejuvenation a more controlled software modernization technique. Code refactoring allows for early validation, and can be used to simplify the extraction process.

In future work we would like to develop more sophisticated techniques for specifying, applying, and validating case-specific refactoring operations, to allow for easier refactoring of code. We would also like to establish whether refactoring *after* rejuvenation could be beneficial.

Acknowledgments

This research was carried out as part of the Vivace program under the responsibility of ESI (TNO) with Royal Philips as carrying industrial partner. The Vivace program is supported by the Netherlands Organisation for Applied Scientific Research TNO.

We would like to thank Peter Blom and Roel Kolman of Philips for their technical support and their help integrating the modernized code.

[3]https://clang.llvm.org/docs/RefactoringEngine.html

[4]https://github.com/cwi-swat/clair

References

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] R. Khadka, B. V. Batlajery, A. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" in *ICSE'14*, 2014, pp. 36–47.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[4] M. C. Feathers, *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[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 *MoDELS'07*, 2007, pp. 482–497.

[6] A. J. Mooij, G. Eggen, J. Hooman, and H. van Wezep, "Cost-effective industrial software rejuvenation using domain-specific models," in *ICMT'15*, 2015, pp. 66–81.

[7] A. J. Mooij, M. M. Joy, G. Eggen, P. Janson, and A. Rădulescu, "Industrial software rejuvenation using open-source parsers," in *ICMT'16*, 2016, pp. 157–172.

[8] S. Klusener, A. J. Mooij, J. Ketema, and H. van Wezep, "Reducing code duplication by identifying fresh domain abstractions," in *ICSME'18*, 2018, pp. 569–578.

[9] M. Pizka, "Straightening spaghetti-code with refactoring?" in *SERP'04*, 2004, pp. 846–852.

[10] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 633–649, 2014.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017.

[13] D. Dams, A. J. Mooij, P. Kramer, A. Radulescu, and J. Vanhara, "Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code," in *SANER'18*, 2018, pp. 552–556.

[14] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 5–18, 2012.

[15] R. M. Fuhrer, M. Keller, and A. Kiezun, "Advanced refactoring in the Eclipse JDT: past, present, and future," in *WRT'07*, 2007, pp. 30–31.

[16] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *OOPSLA'10*, 2010, pp. 286–301.

[17] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Information & Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[18] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in Linux device drivers," in *EuroSys'06*, 2006, pp. 59–71.

[19] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *SCAM'09*, 2009, pp. 168–177.

[20] R. Aarssen, J. J. Vinju, and T. van der Storm, "Concrete syntax with black box parsers," *The Art, Science, and Engineering of Programming*, vol. 3, no. 3, 2019.

[21] I. D. Baxter, C. W. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *ICSE'04*, 2004, pp. 625–634.

[22] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source transformation in software engineering using the TXL transformation system," *Information & Software Technology*, vol. 44, no. 13, pp. 827–837, 2002.

[23] S. S. Somé and T. Lethbridge, "Enhancing program comprehension with recovered state models," in *IWPC'02*, 2002, pp. 85–93.

[24] M. G. J. van den Brand, A. Serebrenik, and D. van Zeeland, "Extraction of state machines of legacy C code with Cpp2XMI," in *BENEVOL'08*, 2008, pp. 28–30.

[25] W. Said, J. Quante, and R. Koschke, "On state machine mining from embedded control software," in *ICSME'18*, 2018, pp. 138–148.