Reducing Code Duplication by Identifying Fresh Domain Abstractions

Steven Klusener  
ESI (TNO)  
Eindhoven, The Netherlands

Arjan J. Mooij  
ESI (TNO)  
Eindhoven, The Netherlands

Jeroen Ketema  
ESI (TNO)  
Eindhoven, The Netherlands

Hans van Wezep  
Philips Healthcare  
Best, The Netherlands

Abstract—When software components are developed iteratively, code frequently evolves in an inductive manner: a unit (class, method, etc.) is created and is then copied and modified many times. Such development often happens when variation points and, hence, proper domain abstractions are initially unclear. As a result, there may be substantial amounts of code duplication, and the code may be difficult to understand and maintain, warranting a redesign.

We apply a model-based process to semi-automatically redesign an inductively-evolved industrial adapter component written in C++: we use reverse engineering to obtain models of the component, and generate redesigned code from the models.

Based on our experience, we propose to use three models to help recover understanding of inductively-evolved components, and transform the components into redesigned implementations. Guided by a reference design, a component’s code is analyzed and a legacy model is extracted that captures the component’s functionality in a form close to its original structure. The legacy model is then unfolded, creating a flat model which eliminates design decisions by focusing on functionality in terms of external interfaces. Analyzing the variation points of the flat model yields a redesigned model and fresh domain abstractions to be used in the new design of the component.

I. INTRODUCTION

In industry, we often encounter development practices in which a next unit (class, method, etc.) is implemented by copying and modifying a previous one, leading to numerous sibling units. Such inductive evolution typically occurs when variation points and, hence, proper domain abstractions are initially unclear. At some point, functionality will have reached a certain level of completeness, while the design lacks essential properties to ensure maintainability [1].

Inductively evolved components naturally arise in iterative software development [2]. Once the maintenance of these components becomes difficult, it is best to redesign [2]. Such a redesign can be seen as an instance of the grow-and-prune model of [3]. The initial development of the component is loosely managed not to limit growth, but leading to an uncontrolled mitosis implementation [3]. Afterwards, once functionality and variation points are clear, the implementation can be pruned into a governed architecture.

Redesign

It is often a challenge to allocate resources to a redesign project; there are risks involved without immediate value in terms of new features. Nevertheless, in the long run, a redesign generally supports maintenance and development of new features by reducing accidental complexity. Given this observation, we investigate methods that enable the redesign of software components in an industrial setting.

Simple redesigns can often be performed using step-wise refactoring [4]. However, if an existing design cannot be motivated other than by historical means, a step-wise approach is often insufficient, and a major redesign may be required. In this context, redesign processes are commonly called modernizations [5], rejuvenations [6], or renovations [7], and are usually associated with legacy software. However, inductively evolved components are not necessarily old; below we use the term legacy code simply to refer to the original code.

An important challenge is to avoid a redesign that closely mirrors the legacy implementation. A good redesign requires moving away from (arbitrary) design decisions encoded in the legacy code, and identifying fresh domain abstractions. To this end, we propose to first recover the functionality that needs to be preserved and describe it at the right level of abstraction.

Recovery can be achieved through reverse engineering [1]. Unfortunately, when the code is the main source of information, recovery can be hard [8]. However, leaving the old code behind enables redesigns with big changes in class hierarchies [9], code size, and use of design patterns [10].

Redesign Process

In our experience, the legacy components that need attention are known to developers. However, by the very nature of these components, the developers’ understanding of them is limited, and often there is no clear vision for a redesign.

To carry out a redesign in a controlled and manageable manner, we use a model-based process (see Figure [1]). The process is based on a reference design, i.e., a high-level design for the type of component considered. The reference design is used both to regain understanding of the legacy code and to create the redesign. However, we do not assume that the legacy code is based on the reference design.

To structure the process, and based on our experiences, we propose to use three intermediate models, summarized in Table [1]. The models are obtained and processed as follows:

1) The legacy code is analyzed. Guided by the reference design, unit specific code fragments are extracted into a legacy model, and reusable code fragments are stored.

2) The legacy model, consisting of extracted fragments in terms of the old design, is unfolded into a flat model. The
Our industrial case study focuses on an adapter component of a control module. The control module is part of a large embedded system. The components of the module interact using Microsoft’s COM (Component Object Model) technology, and, hence, are platform dependent. A long term goal is to redesign the complete control module to reduce code size, accidental complexity, and platform dependence.

Adapters are conceptually simple, but in industrial practice there are often many adapters. Their implementations can also be rather complex due to the various interface technologies involved. These characteristics make adapters a relevant application domain for redesign processes.

The considered adapter has a substantial amount of COM-related interface code that is not isolated from the core functionality. We aim to redesign the adapter in order to reduce code size and accidental complexity, and to ease replacing COM technology with a platform independent solution.

As our goal is to isolate COM-related code in separate layers, our work is related to [6]. However, while [6] focuses on detecting mismatches and other peculiarities in COM-related glue layers, we focus on redesigning a component.

### A. Accidental Complexity

The considered adapter component consists of 21 thousand lines of C++ code distributed over 128 classes and 258 files. The component was developed 5–6 years ago by 4 developers, who no longer work on the component. The component is considered to be difficult to understand, maintain, and extend.

Manual code inspection identified several kinds of accidental complexity:

- Substantial amounts of code duplication as a consequence of inductive evolution and little use of libraries, in particular little use of libraries for common data conversions.
- Over-enthusiastic use of design patterns that make it difficult to understand the functionality.
- Almost 400 occurrences of very verbose COM interactions occurring in 79 of the 128 classes.

Triggered by the above observations, we measured code duplication with two often used tools:

- PMD[^1] with a standard clone length of 100 tokens, reports 22 clones leading to a code duplication ratio of 4%.
- Simian[^2] with a standard clone length of 10 lines, reports 138 clones leading to a code duplication ratio of 14%.

[^1]: http://pmd.github.io/
[^2]: http://www.harukizaemon.com/simian/

---

**TABLE I**

<table>
<thead>
<tr>
<th>Model</th>
<th>Focus</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy</td>
<td>Extracted code</td>
<td>Old design</td>
</tr>
<tr>
<td>Flat</td>
<td>Functionality</td>
<td>External interfaces</td>
</tr>
<tr>
<td>Redesigned</td>
<td>Variation points</td>
<td>New design</td>
</tr>
</tbody>
</table>

All the models are domain-specific (and platform-independent) models of the functionality. The models and code are manually inspected once generated, and the redesigned code is validated against the legacy code. The model extraction is based on analysis techniques tailored to a specific scope[^3]. As a rule, when moving from left to right in Figure 1, the models tend to become smaller, while understanding improves.

Observe that the redesign process combines a bottom-up reverse engineering approach (via code extraction) with a top-down approach (via a reference design). The combination maximizes understanding of the code to enable the redesign.

### Research Method and Contributions

We conducted the reported research as an exploratory case study[^4], with applied researchers and software developers collaborating in an industrial environment. The case study concerned the redesign of an industrial adapter component of a large embedded system. The proposed three models were developed and evaluated in the context of the adapter component, which internally consists of seven adapters.

Our main contributions are:

- A redesign process centered around three models, whose aim is to regain understanding while transforming legacy code into a new implementation (Section III).
- An approach to partially automating the redesign process using parsing techniques, pattern matching, model transformation, and similarity detection (Section IV).
- A demonstration of the effectiveness of the redesign process on an industrial legacy component (Section V).

### II. Case Study: An Industrial Adapter

Fig. 1. The redesign process with its reference design and three models

---

[^1]: http://www.harukizaemon.com/simian/
[^2]: http://pmd.github.io/
[^3]: http://www.harukizaemon.com/simian/
[^4]: https://example.com/research-methodology
The owner of the component also uses PMD to measure code duplication. Unfortunately, the duplication ratios do not reflect the actual amount of duplication we found. This is mostly due to the tools not being able to identify Type-3 clones (identical fragments modulo different subfragments) \[15\]. Hence, we consider the tools not very suitable in support of a redesign.

B. Adapter Mappings and COM Technology

The adapter component connects a Frontend to a Backend. The Backend sends and receives XML data via a Remote Procedure Call (RPC) interface. Communication with the Frontend is COM-based, and uses a library with custom implementations of two common design patterns \[10\]:

- the Command pattern, whose implementations are referred to as Commands in the code base, and
- the Observer or Publish-Subscribe pattern, whose implementations are referred to as UI-Subjects.

Commands and UI-Subjects are managed by CommandManagers and UI-Models, respectively.

Internally, the considered adapter component consists of seven adapters, each connecting a specific pair of interfaces, for a total of 162 function mappings. We redesigned all mappings. The bulk of the functionality relates to 44 mappings referred to as Executers and 66 mappings referred to as Report Parameters. The Report Parameters encompass more than half the classes: 84 out of 128.

Executers: Each Executer receives XML input from the Backend and executes an associated Command in the Frontend. Figure 2 gives typical XML input (for the StartFunc Executer) and specifies actions that should be performed.

Report Parameters: Each Report Parameter observes multiple UI-Subjects in the Frontend. When the value of a UI-Subject changes, a Report Parameter reports this to the Backend via an XML message. Figure 3 provides an example of a Report Parameter (Step_Duration) observing a UI-Subject of type LONG. The figure gives a typical input event (LongItemChanged(4)), describes the actions to be performed when reporting a change, and specifies the resulting XML send to the Backend. This particular Report Parameter also observes five other UI-Subjects, as reflected by the additional attributes in the resulting XML.

III. THE REDesign PROCESS

We next detail our redesign process, first considering the reference design, before expounding on each of the three models and the redesigned code. We illustrate the process by means of the adapter component introduced above.

A. Reference Design

The reference design describes an ideal high-level design of the type of component being redesigned, and serves two purposes. First, irrespective of the legacy design, the reference design provides a frame of reference that we can use to recover our understanding of the legacy code. Second, the reference design serves as a basis along which we can structure our redesigned code.

Case Study: In the case of an adapter component, we use the reference design of Figure 4. This design generalizes the adapter designs of \[16\].

The layers of the design of Figure 4 are intended to separate the instantiation of function mappings from the core design and domain primitives; they also isolate interface specific code. The layers are as follows:

1) Orchestration: creation of mappings and interfaces;
2) Instantiation: unit-specific mappings;
TABLE II

DEFAULT CLASSIFICATIONS FOR ADAPTER LEGACY CODE

<table>
<thead>
<tr>
<th>Layers</th>
<th>Extract</th>
<th>Store</th>
<th>Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orchestration</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Domain</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3) **Core**: design primitives (reusable by sibling units);
4) **Domain**: domain primitives (types and conversions);
5) **Interface**: interaction with neighboring components.

In inductively-evolved adapter components, the separation between the layers is often not clear. This specifically holds for the code representing the Instantiation and Core layers.

In the case of our concrete adapter component, the Instantiation and Core layers are defined by the 162 mappings of the seven adapters. The Domain layer is represented by the data types and conversion routines relevant to the embedded system, and the Interface layers are represented by the employed COM and RPC technologies.

B. Legacy Model Extraction

The redesign process, as depicted in Figure 1, starts by classifying the legacy code and creating a legacy model. We use the following classification categories:

1) **Extract** code fragments that express parameters of sibling units. The parameters are extracted (in an automated fashion) and added to the legacy model.
2) **Store** code clones that are used by multiple sibling units. The clones are later turned into reusable routines. The discriminating properties of the clones are put in the legacy model, and will be passed as parameters to the reusable routines.
3) **Drop** all non-essential legacy code. This is mainly code that exists as a result of previous design decisions, and includes logging and tracing code, wrappers, class hierarchies, and redundant design patterns.

Each part of a reference design has one or more default associated classification categories, as shown in Table II for our adapter design.

We classify our legacy code incrementally. Initially, we may decide to drop all code fragments, but when later analysis and generation steps demand it—or to improve understanding—we may decide to extract or store additional fragments. With regard to understanding, we often find that we need to extract more fragments than strictly necessary for a redesign (see also Section III-D).

Once we have an initial classification, we manually formalize code patterns for the fragments to be extracted. We use the patterns to automate the extraction of the legacy model. In Section IV-A, we discuss the technology behind the automation.

In our experience, it is quite easy to overlook (essential or accidental) code variation. Hence, the automated process reports upon code fragments that do not match our formalized patterns. To reduce the number of false positives, we also define patterns for typical code fragments that need to be dropped (such as logging statements), allowing us to ignore these automatically.

In the legacy model, we aim for a level of granularity that matches the level at which developers reason about the code. For example, although the code may locally consist of a number of small computation steps, we typically do not represent each individual step. Instead, we symbolically evaluate the steps to keep the extracted model manageable and understandable. For example, when we encounter $y = 2 \times x$ followed by $y = y + 1$ in the legacy code, we oftentimes represent this in the model by $y = 2 \times x + 1$.

**Case Study**: Figure 5 presents the legacy model extracted for the StartFunc Executer of Figure 2. The first block of Figure 5 describes an invocation of the initialization method described in the second block. The third parameter in the invocation refers to the CommandManager managing the Command executed by StartFunc. In the legacy code, a pointer to the CommandManager class is passed; here we simply provide sufficient identifying information to obtain a pointer. The second block defines two attributes (manager and command) and invokes the constructor described in the third block, which defines four more attributes.

The execute attribute in the third block of Figure 5 represents the code of Figure 6; note that the representation is very compact compared to the code. Figure 6 contains the two main methods of StartFunc. After having stored the parameter lCookie in the class member m_llCookie at line 15, the first method calls the second at line 16. During model extraction, we inline the call to avoid distinguishing between the methods in the created model, which matches the way the developers reason about the code.

We handle the code of Figure 6 as follows during extraction:

1) Lines 3–5 obtain the value of AcquisitionChannel
1 void CCmdStartFunc::deserializeAndExecute()
2 { } XmlNode& rNode, LONGLONG lCookie)
3 { } XMLNode& rNode, LONGLONG lCookie)
4 XmlNode& rNode, LONGLONG lCookie)
5 void CCmdStartFunc::deserializeAndExecute()
6 { } XmlNode& rNode, LONGLONG lCookie)
7 { } XMLNode& rNode, LONGLONG lCookie)
8 XmlNode& rNode, LONGLONG lCookie)
9 void CCmdStartFunc::deserializeAndExecute()
10 { } XmlNode& rNode, LONGLONG lCookie)
11 void CCmdStartFunc::deserializeAndExecute()
12 { } XmlNode& rNode, LONGLONG lCookie)
13 void CCmdStartFunc::deserializeAndExecute()
14 void CCmdStartFunc::deserializeAndExecute()
15 void CCmdStartFunc::deserializeAndExecute()
16 void CCmdStartFunc::deserializeAndExecute()
17 void CCmdStartFunc::deserializeAndExecute()
18 void CCmdStartFunc::deserializeAndExecute()
19 void CCmdStartFunc::deserializeAndExecute()
20 void CCmdStartFunc::deserializeAndExecute()
21 void CCmdStartFunc::deserializeAndExecute()
22 void CCmdStartFunc::deserializeAndExecute()
23 void CCmdStartFunc::deserializeAndExecute()
24 void CCmdStartFunc::deserializeAndExecute()
25 void CCmdStartFunc::deserializeAndExecute()
26 void CCmdStartFunc::deserializeAndExecute()
27 void CCmdStartFunc::deserializeAndExecute()
28 void CCmdStartFunc::deserializeAndExecute()
29 void CCmdStartFunc::deserializeAndExecute()
30 void CCmdStartFunc::deserializeAndExecute()
31 void CCmdStartFunc::deserializeAndExecute()
32 void CCmdStartFunc::deserializeAndExecute()
33 void CCmdStartFunc::deserializeAndExecute()
34 void CCmdStartFunc::deserializeAndExecute()
35 void CCmdStartFunc::deserializeAndExecute()
36 void CCmdStartFunc::deserializeAndExecute()
37 void CCmdStartFunc::deserializeAndExecute()
38 void CCmdStartFunc::deserializeAndExecute()
39 void CCmdStartFunc::deserializeAndExecute()
40 void CCmdStartFunc::deserializeAndExecute()
41 void CCmdStartFunc::deserializeAndExecute()
42 void CCmdStartFunc::deserializeAndExecute()
43 void CCmdStartFunc::deserializeAndExecute()
44 void CCmdStartFunc::deserializeAndExecute()
45 void CCmdStartFunc::deserializeAndExecute()
46 void CCmdStartFunc::deserializeAndExecute()
47 void CCmdStartFunc::deserializeAndExecute()
48 void CCmdStartFunc::deserializeAndExecute()
49 void CCmdStartFunc::deserializeAndExecute()
50 void CCmdStartFunc::deserializeAndExecute()
51 void CCmdStartFunc::deserializeAndExecute()
52 void CCmdStartFunc::deserializeAndExecute()
53 void CCmdStartFunc::deserializeAndExecute()
54 void CCmdStartFunc::deserializeAndExecute()
55 void CCmdStartFunc::deserializeAndExecute()
56 void CCmdStartFunc::deserializeAndExecute()
57 void CCmdStartFunc::deserializeAndExecute()
58 void CCmdStartFunc::deserializeAndExecute()
59 void CCmdStartFunc::deserializeAndExecute()
60 void CCmdStartFunc::deserializeAndExecute()
61 void CCmdStartFunc::deserializeAndExecute()
62 void CCmdStartFunc::deserializeAndExecute()
Following steps:

1) In the flat model, the manager field always has two attributes: componentId and methodCall, with the method name in the call attribute always being GetCommandManager. Hence, we drop the method name (leaving the interface name).

2) In addition to the above, there is a one-to-one correspondence between interface names and componentId values. Hence, we apply abstraction. We do this by splitting off combinations of componentId values and interface names into a separate model concept (an example of which is depicted in the first block of Figure 8).

3) The correspondence between prefixes of command values and componentId values is also one-to-one. Hence, we also add the prefix (CommandIDASC in the case of StartFunc) to the model concept introduced in step (2).

4) The property attribute is always identical to the function attribute, with the former being a left-over of an old design decision. Hence, we drop the property attribute.

5) The combination of the function and channel attributes is unique, and forms a logical key. Hence, we use the combination to identify Executers (such as StartFunc).

6) If a cookie is processed, then this is always done in the same manner. Hence, we store the details and represent the processing by the clause **pass** COOKIE.

7) If an attribute must be passed, such as AcquisitionChannel in the case of StartFunc, then there are only two relevant parameters: attribute name and type. In the redesigned model we make this explicit by omitting all other parameters. We do note that the other parameters aided us while trying to understand the functionality.

In Section 4-E, we discuss an automated technique that supports identifying the above variation points and implied abstractions.

Observe that the structure of the redesigned model of Figure 8 differs significantly from that of the legacy model of Figure 5. In fact, the differences are even greater, as another major restructuring occurred not visible in the figure: in the legacy code (and, hence, in the legacy and flat models) some adapter mappings were grouped by Backend interface, whereas in the redesign (and the redesigned model) we grouped them by Frontend interface. This led to a reduction in the number of model concepts, allowing for more reuse.

For the Report Parameter mappings (not shown), the flat model had become very detailed in order to capture all functionality (not in the least because of the large number of UI-Subjects associated with each Report Parameter). The redesigned model helped to improve understanding of the few essential variation points, which in turn helped to generate better code. We also found a few inconsistencies in data conversions. In particular, the processing of initial values sometimes differed subtly from the processing of subsequent values. In practice, these inconsistencies were immaterial, and we uniformized the conversion routines during the redesign.

### E. Code Generation and Code Construction

The new software design is based on the reference design and the domain abstractions identified during construction of the redesigned model. The redesigned model is used to generate the code for the variation points, while all one-of-a-kind code is either based on stored code fragments, or manually redeveloped.

**Case Study:** In the case of our industrial adapter component, we constructed and generated the various layers as follows:

- **Orchestration.** During extraction of the legacy model we stored the fragments responsible for orchestration. We reused these fragments to construct the Orchestration layer.

- **Instantiation.** For each of the 7 adapters, we generated a class that defines its mappings. In the case of StartFunc, we generated the code of Figure 9 from the second block of Figure 8. The structure of the code clearly resembles that of the model, and, as such, we can consider the Instantiation layer to use an embedded or internal DSL [17].

- **Core.** This layer was constructed manually based on the new domain abstractions. The class hierarchy used in the layer differs significantly from the legacy hierarchy.

- **Domain.** This layer consists of the data conversions that were stored during model extraction.

- **Interface.** Like the Domain layer, this layer mostly consists of stored functionality. We also generated part of

---

**Fig. 8.** The redesigned model of StartFunc

**AddExecuter(**
CExecuter::CreateInstance('StartFunc',
ALLCHANNELS,
APPLICATION_STATE_CONTROLLER),
CommandIDASC::StartFuncCmd,
CExecuter::CreateInstance( 'AcquisitionChannel'*);
CExecuter::CreateInstance( 'AcquisitionChannel' ));

**Fig. 9.** The code generated for StartFunc
the layer to enable accessing the Frontend in a uniform manner. For example, the code of Figure 10 was generated from the first block of Figure 8. The code is used to acquire a pointer to the CommandManager referenced by StartFunc.

In addition to code, we also generated documentation describing all mappings and their properties in tabular form. The documentation served as a basis for communication with stakeholders.

F. Validation

The models generated during the redesign process are inherently incomplete, as they focus on distinguishing features. Although we do complement the models with stored code fragments, even this does not fully capture the functionality: some functionality is dropped during extraction only to be redeveloped later. As a result, it is next to impossible to formally prove the correctness of the redesign with respect to the legacy implementation \[13, 19\]. The situation is further complicated if we purposefully deviate from the original functionality in order to make the redesign more uniform, e.g., by removing inconsistencies between clones \[20\].

Given the above, our validation strategy is two-fold:

1) inspect each model once generated (early fault detection);
2) test the redesigned implementation (final check).

For model inspection it is important that the models are human readable. For testing we use (and refine) the existing test suites of the legacy implementation. Alternatively, we could have used the approach from \[21\], which applies model learning and equivalence checking to the legacy and redesigned implementations.

The expected quality of the validation influences the amount of risk we may take during the redesign. For example, by ignoring (apparently irrelevant) code during extraction, the redesign may proceed faster, but we need to be able to verify that the code was indeed irrelevant (which requires high-quality validation).

Case Study: During the redesign of the adapter component, we manually inspected each of the generated models after each iteration. This provided valuable feedback, and led to many improvements in the models, especially during early iterations.

With regard to testing, the adapter component, as part of a much larger control module, can currently only be tested as part of that module. The test suite of the module consists of around a thousand test cases, which generate about 18 million lines of log data. Besides validating the redesigned component by exercising the test suite, we also compared the log data of the legacy and redesigned implementations (see also Section IV-D). This comparison was instrumental in the discovery of a number of issues with early versions of the redesigned component.

The final version of redesigned component passed all tests of the legacy component (when embedded in the much larger control module), and was transferred to the responsible development team.

IV. AUTOMATION IN THE REDESIGN PROCESS

We next discuss the automation we used in our case study.

A. Processing Industrial Code

To enable automatic extraction of a legacy model, we parse the legacy code. In our case study, we used Eclipse CDT’s C++ parser, following the approach of \[6\].

Processing C++ parse trees is rather involved and has a steep learning curve. To ease model extraction, we have implemented a small library that handles common extraction operations. The library includes operations such as extracting function names, return types, and function parameters.

The library also provides a pattern matching facility, which we used during the pattern formalization and model extraction process discussed in Section II-B. The patterns are written using concrete syntax (see Figure 11 for an example, where identifiers starting with $ denote placeholders). The library supports both exact matches and iterating through lists (of statements) to search for occurrences of patterns.

B. Model Transformation Technology

The intermediate results of our redesign process are models obtained through model transformation. To effectively develop
and perform the model transformations, we require the following from the model technology we use to automate them:

- Each transformation should be runnable and debuggable in isolation. Hence, the models should be machine-processable.
- It should be possible to validate intermediate results. Hence, the models should have a human readable representation.
- It should be easy to construct code generators.

In our case study, we used textual models that are both machine-processable and human readable. Each model was based on its own domain-specific language (DSL), which we developed using Xtext\[4\].

All model-to-model transformations were implemented as model-to-text transformations in Xtend\[4\] in order to avoid building large expression trees in terms of abstract syntax. A drawback to this approach is that syntax definitions are duplicated: each definition is encoded both implicitly in the text generation process of one transformation, and explicitly in the parser of the subsequent transformation.

Alternatively, we could have used model-to-model transformations, and relied on serializers and formatters to obtain human readable representations, e.g., using the MoDisco framework\[22\]. Unfortunately, MoDisco currently does not support C++. Tools such as Rascal\[23\], which combine concrete and abstract syntax, might offer an alternative approach: they avoid the dichotomy between model-to-text and model-to-model transformations.

C. Similarity Detection

During model extraction and refolding, we look for natural patterns and abstractions. To find these, we experimented with a similarity analysis based on token-frequency vectors. It is not yet clear how generic this technique is, but initial results are promising. The technique is able to detect Type-3 code clones—contrary to the tools explored in Section II-A—and is mostly language independent. We give a brief overview of the technique.

The technique starts by splitting the code, or model, into fragments. In the case of code, the fragments are usually files or methods. In the case of models, they are usually modeling constructs used for grouping, e.g., the construct describing StartFunc in Figure 7. Next, each fragment is split into tokens, and tokens representing punctuation (such as braces, semicolons, etc.) are dropped. The technique is now applied in one of two ways to the remaining tokens:

- **Token similarity.** For each token \( x \) we construct a token-frequency vector counting for every token \( y \) the number of fragments containing both \( x \) and \( y \). We use the vectors to analyze which tokens always occur together, and, hence, are candidates to become (parameterized) model concepts.

- **Fragment similarity.** For each fragment \( F \) we construct a token-frequency vector counting for every token \( y \) the number of occurrences of \( y \) in \( F \). We use the vectors to analyze which fragments are similar, and, hence, are candidates to become (parameterized) model concepts.

We perform two operations on the token-frequency vectors:

- **Similarity computation.** The similarity between two token-frequency vectors is expressed as a ratio between 0 and 1. We compute the ratio as the cosine similarity of the vectors; ratios above 0.8 are typically interesting. Based on the ratios we draw token-similarity diagrams using hierarchical cluster analysis.

- **Discriminator computation.** Given two token-frequency vectors with high similarity, the difference between them is a vector of discriminating tokens. If a model concept is introduced based on the highly similar vectors, the discriminating tokens are likely to become parameters.

An example of a token-similarity diagram can be found in Figure 12. The diagram is computed from our case study’s flat model (of which Figure 7 is part). The value of 1.00 indicates that the interface IASCSetCHN always occurs in combination with the method SetCHN. This suggests that the interface and method may be combined, as we actually did in Section III-D.

The method StringToCHN has a similarity value of 0.92 when compared to the other two tokens of Figure 12. Further investigation showed that the method StringToCHN also occurs in combination with another interface (not shown).

We also applied the similarity detection technique to the methods occurring in the legacy code base. This exposed significant similarity between the deserializedAndExecute and executeUICommand methods of the StartFunc Executer (as depicted in Figure 6), and the sibling methods of all other Executers. The discriminators revealed precisely the parameters we extracted from the methods in Section III-B.

Because our technique is rather coarse, it gives rise to far less false negatives than the tools explored in Section II-A. On the other hand, our technique may also give rise to more false positives, because the structure of the code or model is ignored. However, in practice, this latter effect seems to be limited, especially when small fragments are excluded from the analysis.

The idea to detect similarity between code fragments using vectors of tokens is not new\[24, 25\]. However, our use of the cosine similarity to compute similarity ratios leads to a simpler technique.

---

**Fig. 12.** A token-similarity diagram

- IASCSetCHN
- SetCHN
- StringToCHN

- 0.92
- 1.00

\[\text{http://www.eclipse.org/Xtext/}\]

\[\text{http://www.eclipse.org/xtend/}\]
TABLE III
CODE METRICS FOR THE INDUSTRIAL ADAPTER

<table>
<thead>
<tr>
<th>Layer</th>
<th>Files</th>
<th>LoC</th>
<th>LoC%</th>
<th>Gen</th>
<th>Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy total</td>
<td>258</td>
<td>21288</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Orchestration</td>
<td>6</td>
<td>282</td>
<td>4%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Instantiation</td>
<td>14</td>
<td>1616</td>
<td>21%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Core</td>
<td>47</td>
<td>2572</td>
<td>33%</td>
<td>0%</td>
<td>20%</td>
</tr>
<tr>
<td>Domain</td>
<td>6</td>
<td>639</td>
<td>8%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>COM Interface</td>
<td>32</td>
<td>2426</td>
<td>31%</td>
<td>41%</td>
<td>40%</td>
</tr>
<tr>
<td>XML Interface</td>
<td>7</td>
<td>276</td>
<td>4%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Redesign total</td>
<td>112</td>
<td>7811</td>
<td>100%</td>
<td>33%</td>
<td>34%</td>
</tr>
</tbody>
</table>

D. Test Suite Refinement Using Log Data

Before embarking on the redesign of a component, we perform a semi-automatic analysis of the log data produced while running the test suite (of either the component, or of some larger module it is part of). From the log data we extract input/output-pairs describing the behavior of the component. If these pairs cannot be extracted, we extend the legacy implementation to generate them.

Running the test suite against the legacy component, we obtain reference input/output-pairs. The pairs allow us to assess test coverage and, consequently, allow us to determine the risks that can be taken during a redesign (see also Section III-F). Availability of the pairs also offers the following opportunities:

- **Refine existing test suites** by comparing input/output-pairs generated during testing with the reference pairs; this may reveal errors even if testing succeeds. The comparison may not be entirely trivial as we may have to abstract from certain differences, e.g., those due to thread or process scheduling.
- **Create unit tests** based on the reference pairs (if unit tests do not yet exist). Each pair can be turned into a test case.
- **Improve test coverage** based on input/output-pairs that are missing from the reference set.

Note that the idea to exploit log data for testing is not new and has been explored elsewhere [26].

V. CASE STUDY: EFFECTIVENESS OF THE REDESIGN

Table III presents code metrics for both the legacy and redesigned implementations of the industrial adapter from our case study. The metrics for the redesigned implementation are broken down per layer of the adapter reference design. Since the source files of the legacy implementation often relate to multiple layers, we do not provide a similar breakdown for this implementation: any division would be arbitrary.

For both implementations, Table III indicates the number of files, and lines of code (LoC)—where a line of code is defined as any non-empty, non-comment line that does not consist entirely of punctuation. For the redesigned implementation the table also indicates the relative size of each layer with respect to total component size (LoC%). The column Gen (Generated) indicates the percentage of LoC generated from the redesigned model. The column Stored estimates the percentage of LoC derived from stored code fragments.

Our redesign reduced code size by a factor of almost 3. As already touched upon in Section II-A, using PMD or Simian to guide a step-wise refactoring would most likely not have resulted in a similar code reduction: PMD and Simian only report 4% and 14% code duplication for the legacy implementation, respectively.

The redesign process led to the identification of fresh domain abstractions and primitives, which we implemented manually, encompassing 80% of the Core layer and 19% of the COM Interface layer, or, in total, 33% of the redesigned implementation. Although significant in terms of the redesigned implementation, this only amounts to 12% of the original code base, with similar amounts having been generated and derived from stored code fragments.

**Change Scenarios:** In the redesigned implementation we can deal with typical changes in the following ways:

- **Add/modify an adapter mapping.** Adding or modifying a mapping affects only a single defining clause in the Instantiation layer. Hence, we estimate that a mapping can be added or changed within an hour. Before, an addition or change could easily take in excess of two days.
- **Add a new mapping type.** Adding a mapping type is not likely to be required, as a large number of types is already supported. However, if such a change does need to be made, some complexity is involved, as it most likely affects both the Core and Instantiation layers.
- **Change a common implementation pattern.** Similar to adding a mapping type, changing an implementation pattern is not likely to be required. However, if such a change does need to be made, it will only affect a single layer (i.e., the Core, Domain, or Interface layer), as each individual pattern is contained in a single layer.
- **Replace COM technology.** Replacing COM technology will affect the COM Interface layer only; other layers can be left unchanged. Originally, most of the code would need to be touched.

Given the above observations, we can consider the deployed architecture [27] of the redesigned component to consist of the Core layer, as it is the only layer that is hard to change. This is in stark contrast to the legacy implementation, which was hard to change overall. Hence, besides substantial code reduction, we also significantly reduced the size of the deployed architecture.

VI. THREATS TO VALIDITY

Threats to **internal** validity come from the way in which we carried out our case study. Although we employed the legacy and flat models from the start based on earlier experiences, the redesigned model was only introduced after several iterations—initial versions of the process generated code directly from the flat model. Although this is the case, we did see substantial improvements in quality of the generated code once we introduced the redesigned model and, hence, consider the redesigned model to be important.

Threats to **construct** validity come from the way in which we evaluated our case study. In particular, the considered
change scenarios and associated time estimates do not derive from actual observations. We do expect the change scenarios and time estimates to be accurate, as they were established as part of discussions with stakeholders.

Threats to external validity come from the way in which the results will generalize to other software components. We only considered seven different (but somewhat related) adapters, and plan to perform further case studies to establish whether our techniques generalize to other software components.

VII. CONCLUSIONS AND FUTURE WORK

In iterative software development, code often evolves inductively by copying and modifying earlier developed units. Initially this may be the way forward, but once functionality and variation points become clear, the code should be redesigned to ensure maintainability. In our experience, many industrial software components have substantial amounts of code duplication due to inductive evolution.

We proposed a semi-automated redesign process that focuses on regaining human understanding and identifying fresh domain abstractions. The process combines a top-down approach based on a reference design, with a bottom-up approach based on transformations of legacy code into a redesigned implementation via three models. The reference design is used to understand the legacy code, to steer the model extraction, and to guide the new design.

The three proposed models are used to ensure that previous design decisions are forgotten before new decisions are made. This helps to ensure that the new design does not resemble the old one. Each subsequent model is more compact than the previous one, reflecting an increasing understanding of the essential functionality and variation points.

We illustrated the effectiveness of the redesign process by presenting an industrial case study concerned with the redesign of an adapter component. Using the process, we reduced the size of the component by a factor of almost 3. About 12% of the original code was carried over to the new implementation, and a similar amount of code was manually redeveloped. Due to proper layering of the redesigned component, we estimate that future changes will take significantly less effort. In particular, the code related to COM interface technology has been isolated, which facilitates easy replacement.

As inductive evolution and adapter components are omnipresent in industrial practice, we expect that numerous of legacy components could benefit from our approach. In each case, we will need to select or define an appropriate reference design and appropriate domain-specific modeling languages.

As future work, we plan to further evaluate our redesign process while redesigning other industrial software components. We also intend to study other techniques for discovering code and model patterns that could be used to identify abstractions during the extraction and refolding steps.

REFERENCES


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


