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 I. 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

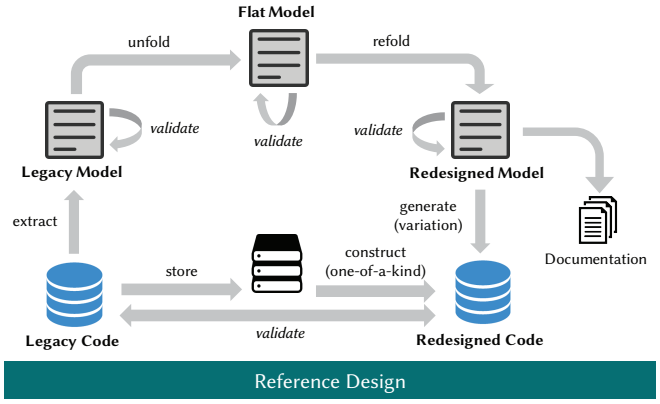

Fig. 1. The redesign process with its reference design and three models

TABLE I
THE THREE MODELS AND THEIR ROLES

Model	Focus	Primitives
Legacy	Extracted code	Old design
Flat	Functionality	External interfaces
Redesigned	Variation points	New design

flat model is design independent and defines the component's functionality in terms of external interfaces.

3) The flat model is refolded into a *redesigned model* by uncovering the variation points of the flat model. As a result, fresh domain abstractions for a new design are also obtained.
4) New code is constructed based on the reference design and fresh domain abstractions. The redesigned model is used to generate code for the variation points, and the code fragments stored in the first step are used to implement the one-of-a-kind functionality.

All the models are domain-specific (and platform-independent [11], [12]) 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 [13]. 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) [1]. 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 [14], 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

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/

XML input:

```
<StartFunc Channel="ALLCHANNELS">
  <AcquisitionChannel>FRONTAL</AcquisitionChannel>
</StartFunc>
```

Actions:

1) Extract the `AcquisitionChannel` value.
2) Convert the value to a dedicated data type.
3) Pass the value to the associated Command.
4) Pass a cookie (provided as a separate parameter in the incoming RPC call) to the Command.
5) Lock, execute, and unlock the Command.

Fig. 2. The `StartFunc` Executer function

Input event:

```
LongItemChanged(4)
```

Actions:

1) Store the parameter value of the event.
2) Collect the most recent value of each related UI-Subject, and apply any required data conversions.
3) Create and send an XML message with all values.

XML output:

```
<Step_Duration Status="ENABLED" Channel="ALLCHANNELS"
  ValueStatus="KNOWN" ValueChanging="false">
 <Duration Min="1" Max="20" StepSize="1">4</Duration>
</Step_Duration>
```

Fig. 3. The `Step_Duration` Report Parameter function

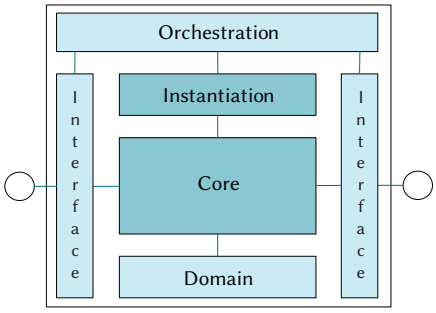

Fig. 4. Adapter component reference design

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

Layers	Extract	Store	Drop
Orchestration		✓	
Instantiation	✓		
Core		✓	✓
Domain		✓	
Interface		✓	

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.

```
invoke : CreateStartFunc(
  "StartFunc",
  ALLCHANNELS,
  [ componentId : CLSID_COASCAppStateController,
    methodCall  : IASC.GetCommandManager] )

-----------------------------------------------

CreateStartFunc(Name, Relevance, CmdManager) ->
  manager : CmdManager
  command : CommandIDASC.StartFuncCmd
  invoke  : CCmdStartFunc(Name, "StartFunc", Relevance)

-----------------------------------------------

CCmdStartFunc(FuncName, PropName, Relevance) ->
  function : FuncName
  property : PropName
  channel  : Relevance
  execute  : extract "AcquisitionChannel" :
               apply StringToCHN yielding CHN
               set IASCSetCHN.SetCHN
             extract COOKIE :
               set IASCCommandCookie.SetCookie
```

Fig. 5. The legacy model of `StartFunc`

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 * x` followed by `y = y + 1` in the legacy code, we oftentimes represent this in the model by `y = 2 * 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_lCookie` 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   // Get "AcquisitionChannel" string value from the input
 4   CString sChannel =
 5       rNode.GetElement("AcquisitionChannel")->GetStringValue();
 6   // Convert string to CHN, store value
 7   if (sChannel.CompareNoCase("FRONTAL") == 0) {
 8     m_eChannel = CHN_FRONTAL;
 9   } else if (sChannel.CompareNoCase("LATERAL") == 0) {
10     m_eChannel = CHN_LATERAL;
11   } else {
12     m_eChannel = CHN_BIPLANE;
13   }
14   // Store cookie
15   m_llCookie = lCookie;
16   executeUICommand();
17 }
18
19 void CCmdStartFunc::executeUICommand() {
20   // Get associated Command
21   IInfraCommand *pInfCmd = GetUICommand();
22   if (pInfCmd != NULL) {
23     bool bResult =
24         Lock(pInfCmd, CCmdPerformFunctionCommand::v_unPriority);
25     if (bResult) {
26       IASCSetCHN* pSetCHN = NULL;
27       HRESULT hr = pInfCmd->QueryInterface(
28           INFRA_IID_PPVOID(IASCSetCHN, &pSetCHN));
29
30       if (SUCCEEDED(hr)) {
31         // Pass CHN value to Frontend
32         if (pSetCHN != NULL) {
33           hr = pSetCHN->SetCHN(m_eChannel);
34
35           if (SUCCEEDED(hr)) {
36             IASCCommandCookie* pIASCCommandCookie = NULL;
37             HRESULT hr = pInfCmd->QueryInterface(
38                 INFRA_IID_PPVOID(IASCCommandCookie, &pIASCCommandCookie));
39
40             if (SUCCEEDED(hr)) {
41               // Pass cookie value to Frontend
42               if (pIASCCommandCookie != NULL) {
43                 hr = pIASCCommandCookie->SetCookie(m_llCookie);
44
45                 // Execute associated Command
46                 if (SUCCEEDED(hr)) { HRESULT hRes = pInfCmd->Execute(); }
47                 pIASCCommandCookie->Release();
48                 pIASCCommandCookie = NULL;
49               }
50             }
51           }
52           pSetCHN->Release();
53           pSetCHN = NULL;
54         } else {
55           hr = E_FAIL;
56         }
57       }
58       bResult = Unlock(pInfCmd);
59     }
60     CGenUtilities::releaseAndNull(pInfCmd);
61   }
62 }
```

Fig. 6. The two main methods of `StartFunc`

from the provided XML. We extract `AcquisitionChannel` and add it as a parameter to the legacy model; the parameter belongs to the Instantiation layer. The remainder of the fragment is stored, to become part of the Interface layer.

2) Lines 6–13 convert a string to an element of type `CHN`. We store the fragment; the conversion belongs to the Domain layer. The type `CHN` is specific to `StartFunc` and is put into the legacy model together with a reference to the conversion.
3) Lines 26–33 pass the converted value to a Command. The code is COM-specific and belongs to the Interface layer. We store the fragment together with lines 52–53 (responsible for clean-up). The `IASCSetCHN` and `SetCHN` values are specific to `StartFunc` and are put into the legacy model.
4) Lines 36–43 pass a cookie to the Command. The code is again COM-specific and belongs to the Interface layer. We store the fragment together with lines 47–48 (responsible for clean-up). The `IASCCommandCookie` and `SetCookie` values are specific to `StartFunc` and are put into the legacy model.
5) Lines 20–25, 45–46, and 58–60 lock, execute, and unlock the Command. This functionality belongs to the Core layer; we store the fragments.

```
manager  :
  componentId : CLSID_COASCAppStateController
  methodCall  : IASC.GetCommandManager
command  : CommandIDASC.StartFuncCmd
function : "StartFunc"
property : "StartFunc"
channel  : ALLCHANNELS
execute  : extract "AcquisitionChannel" :
             apply StringToCHN yielding CHN
             set IASCSetCHN.SetCHN
           extract COOKIE :
             set IASCCommandCookie.SetCookie
```

Fig. 7. The flat model of `StartFunc`

Note that the Interface related fragments (lines 26–33 and 36–43) are similar, but differ slightly with respect to error handling (the assignment at line 55 technically belongs to the first fragment). Manual analysis revealed that the difference is accidental and that the code at line 55 can be dropped, making the fragments homogeneous, and enabling us to reduce the amount of stored code.

Although the adapter component was implemented manually, the amount of incidental variation among the Executers is limited due to their inductive evolution. All Executers are structured similarly, and parameters are easily extracted.

C. Model Unfolding

In order to gain insight in the legacy code's functionality, the next step is to eliminate old design decisions. To this end, we unfold the legacy model into a *flat model*. In practice, this involves removing design patterns, inlining methods, and simplifying data- and control-flow. We use automation to avoid mistakes. In contrast to the local symbolic evaluation applied during model extraction, this step focuses on global reductions.

Case Study: The three blocks of the legacy model of Figure 5 form a call chain. The first block provides a number of parameters and invokes the second block. In turn, the second block is similar and invokes the third block. We eliminate the call chain to obtain the flat model of Figure 7.

D. Model Refolding

In the refolding step, we analyze the flat model and introduce fresh domain abstractions. We look for model fragments that occur multiple times, or depend on each other. In general, we strive for compactness (not as an independent goal, but as a guideline) to avoid code duplication later on.

In this step it may also become clear that the legacy and flat models contain more information than strictly required for a redesign (although the additional information may have been used to regain understanding). Unexpected inconsistencies in the functionality may also be revealed during this step, and may be manually repaired as part of the refolding.

```
APPLICATION_STATE_CONTROLLER :
  identifier     : CLSID_COASCAppStateController
  interface      : IASC
  command prefix : CommandIDASC

-------------------------------------------------

"StartFunc" / ALLCHANNELS ->
  component      : APPLICATION_STATE_CONTROLLER
  command postfix : StartFuncCmd
  execute        : pass "AcquisitionChannel"<CHN>
                   pass COOKIE
```

Fig. 8. The redesigned model of `StartFunc`

```
AddExecuter(
  CExecuter::CreateInstance(
    "StartFunc",
    ALLCHANNELS,
    APPLICATION_STATE_CONTROLLER),
  CommandIDASC::StartFuncCmd,
  CExecSetValueCmdXML<CHN>::CreateInstance(
    "AcquisitionChannel"),
  CExecSetValueCmdCookie::CreateInstance());
```

Fig. 9. The code generated for `StartFunc`

Case Study: For `StartFunc` we obtain the redesigned model of Figure 8 from the flat model of Figure 7 via the 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 IV-C, 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

```
IASC*
InterfaceProxy::GetAppStateController() {
  if (m_pASCAppStateController == nullptr) {
    DO_FUNC(::CoCreateInstance(
      CLSID_COAppStateController,
      nullptr,
      CLSCTX_LOCAL_SERVER,
      INFRA_IID_PPVOID(IASC, &m_pAppStateController)));
  }
  return m_pAppStateController;
}

ICommandManager*
InterfaceProxy::GetCmdMngrAppStateController() {
  if (m_pCmdMngrAppStateController == nullptr) {
    DO_FUNC_ALLOW_ERROR(
      GetAppStateController()->GetCommandManager(
        INFRA_IID_PPIUNK(
          ICommandManager,
          &m_pCmdMngrAppStateController)));
    ACCEPT_COM_ERROR(E_NOTIMPL);
    END_FUNC;
  }
  return m_pCmdMngrAppStateController;
}
```

Fig. 10. The code generated for acquiring a pointer to a CommandManager

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 [18], [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).

```
if ($in.CompareNoCase("FRONTAL") == 0) {
  $out = CHN_FRONTAL;
} else if ($in.CompareNoCase("LATERAL") == 0) {
  $out = CHN_LATERAL;
} else {
  $out = CHN_BIPLANE;
}
```

Fig. 11. A code pattern for extracting a data conversion

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 III-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.[3]

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, may be combined into a single model concept.
- *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.

[3]http://www.eclipse.org/Xtext/

[4]http://www.eclipse.org/xtend/

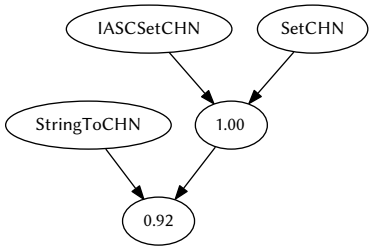

Fig. 12. A token-similarity diagram

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 `deserializeAndExecute` 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 negative*s 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.

TABLE III
CODE METRICS FOR THE INDUSTRIAL ADAPTER

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

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

[1] S. Tilley, S. Paul, and D. Smith, "Towards a framework for program understanding," in *WPC'96*, 1996, pp. 19–28.

[2] V. Basili and A. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. 1, no. 4, pp. 390–396, 1975.

[3] D. Faust and C. Verhoef, "Software product line migration and deployment," *Softw. Pract. Experience*, vol. 33, no. 10, pp. 933–955, 2003.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, 1999.

[5] R. Khadka, P. Shrestha, B. Klein, A. Saeidi, J. Hage, S. Jansen, E. van Dis, and M. Bruntink, "Does software modernization deliver what it aimed for?" in *ICSME'15*, 2015, pp. 477–486.

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

[7] M. van den Brand, A. Sellink, and C. Verhoef, "Generation of components for software renovation factories from context-free grammars," in *WCRE'97*, 1997, pp. 144–153.

[8] A. Eastwood, "It's a hard sell–and hard work too (software reengineering)," *Computing Canada*, vol. 18, p. 35, 1992.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

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

[11] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J. Jézéquel, "Model-driven engineering for software migration in a large industrial context," in *MoDELS'07*, 2007, pp. 482–497.

[12] M. Grieger, M. Fazal-Baqaie, G. Engels, and M. Klenke, "Concept-based engineering of situation-specific migration methods," in *ICSR'16*, 2016, pp. 199–214.

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

[14] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 285–311.

[15] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[16] A. Mooij and M. Voorhoeve, "Specification and generation of adapters for system integration," in *Situation Awareness with Systems of Systems*. Springer, 2013, pp. 173–187.

[17] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. http://dslbook.org/: Online, 2013.

[18] M. Ward, "Pigs from sausages? Reengineering from assembler to C via FermaT transformations," *Sci. Comput. Program.*, vol. 52, no. 1–3, pp. 213–255, 2004.

[19] R. Van Der Straeten, V. Jonckers, and T. Mens, "A formal approach to model refactoring and model refinement," *Software and System Modeling*, vol. 6, no. 2, pp. 139–162, 2007.

[20] S. Wagner, A. Abdulkhaleq, K. Kaya, and A. Paar, "On the relationship of inconsistent software clones and faults: An empirical study," in *SANER'16*, 2016, pp. 79–89.

[21] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of legacy software using model learning and equivalence checking: An industrial experience report," in *IFM'16*, 2016, pp. 311–325.

[22] 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.

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

[24] G. Cosma and M. Joy, "An approach to source-code plagiarism detection and investigation using latent semantic analysis," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 379–394, 2012.

[25] S. Grant and J. Cordy, "Vector space analysis of software clones," in *ICPC'09*, 2009, pp. 233–237.

[26] J. Andrews, "Testing using log file analysis: tools, methods, and issues," in *ASE'98*, 1998, pp. 157–166.

[27] A. Klusener, R. Lämmel, and C. Verhoef, "Architectural modifications to deployed software," *Sci. Comput. Program.*, vol. 54, no. 2–3, pp. 143–211, 2005.