Refactoring Legacy Code using Cleaning Up Cycles: An Experience Report

Andrei V. Girjoaba, Andrea Capiluppi, *Member, IEEE*
Department of Computing Science
University of Groningen (NL)
{a.v.girjoaba,a.capiluppi}@rug.nl

2024 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-9568-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSME58944.2024.00079

***Abstract*—Due to the lack of standards at the time they were written, many codebases suffer from reduced code quality. Some of these systems can not be replaced and need to be maintained to guarantee the healthy development of the software. In the context of this paper, we are presenting a case study performed on the legacy codebase of ASML, a leading Dutch company specializing in the design and manufacture of advanced semiconductor lithography equipment. As new policies were enforced within the company, many specific interfaces were deprecated. The presence of such obsolete interfaces in the codebase adds to the overall complexity of the legacy system, while also permitting a future violation of the Interface Segregation Principle, one of the five SOLID design principles.**

The objective of this paper is to present our experience with refactoring the specified interfaces. Besides performing the clean-up, we were tasked to find an optimal, iterative, and repeatable approach to performing the clean-up of these interfaces while documenting our findings in a reproducible manner for future employees.

To discover an efficient clean-up strategy, we used the Action Research methodology. Going through the *"Planning"*, *"Acting"*, *"Observing"* and *"Reflecting"* phases of this methodology repeatedly, we revised the strategy and executed our refactoring work accordingly.

After multiple refactoring operations, our process stabilized and we introduced the "Cleaning-Up Cycle". We spent less time and had fewer failures per interface as our cycle evolved, despite attempting more complex types of interfaces later.

We found the Clean-Up Cycle to represent a valuable strategy when executing this type of refactoring. This was fed back to the ASML team and considered the foundation for future employees continuing our work.

***Index Terms*—Industrial experience, Refactoring, Legacy code, SOLID principles.**

I. Introduction

Software maintenance remains the most extensive and costly phase of a software system's lifecycle. Ensuring a clean software design is crucial, as it enables software engineers to implement new features more rapidly and reduces the incidence of bugs within the program [1], [2]. Numerous studies have highlighted these benefits across various contexts. For instance, large legacy mainframe systems have demonstrated the importance of maintaining a robust design [3], [4], [5], [6]. Similarly, in mission- [7], [8] and safety-critical [9] systems, the maintenance of clean software design has been shown to be vital. Furthermore, widely adopted open-source systems also benefit significantly from meticulous software maintenance, which impacts their overall sustainability and security [10], [11], [12].

After the term "software aging" was coined in 1994 by David Lorge Parnas [13], indicating software maintainability as a crucial factor for systems evolvability, the notion of technical debt [14] has rapidly become a prevalent aspect in assessing code quality and its life expectancy. Software must respect quality standards to allow developers to keep up with the requirements of a constantly evolving market.

Legacy code, characterized by its evolution over many years or even decades of software development, frequently suffers from a significant accumulation of technical debt. This technical debt arises from various sources, including quick fixes, lack of documentation, suboptimal architectural decisions, and outdated technologies. The process of addressing these issues is often complex and labor-intensive, requiring substantial effort to identify and refactor code smells, update outdated components, and rectify architectural design compromises. This scenario underscores the importance of proactive technical debt management and continuous refactoring practices to maintain the health and agility of the codebase [15], [16].

The experience report reported in this paper is based on legacy code, and its technical debt accumulated over twenty years: our working collaboration with ASML involved the refactoring of a portion of this legacy codebase, and the creation of repeatable refactoring patterns for future use. During three months, we were assigned the task of cleaning-up several deprecated interfaces[1]: those became obsolete when clients were prohibited from writing strongly typed code (e.g., Java, C#, or Swift), although some of their functionality was still required internally. Their presence in the codebase represented a potential risk for violating the Interface Segregation Principle (ISP), as they could still be accessed. The ISP is one of the five SOLID principles [17] and states that "*Clients should not be forced to depend upon interfaces that they do not use*."

We approach the problem of reducing technical debt by developing a tailored solution for the problem at hand. Following the action research (AR) methodology we come up with the Cleaning-Up Cycle. We applied the process of acting, observing, reflecting, and planning described by AR while refactoring multiple interfaces before the introduced solution fully matured.

In undertaking this study, we conducted an extensive review of past research literature to underpin our methodology.

[1]to preserve data anonymity, in this paper, we will refer to these as 'Q interfaces'.

2576-3148/24/$31.00 ©2024 IEEE
DOI 10.1109/ICSME58944.2024.00079

However, our exploration revealed very few documented initiatives focused on refactoring legacy industry code, which constrained the knowledge we could draw upon. As such, the work presented here is a contribution to this scarce research area, and particularly to the *industry-as-lab* approach [18], [19], which involves utilizing real-world industrial settings as experimental environments for research and development purposes [20]. In this approach, researchers and practitioners leverage the operational contexts of industries to conduct practical testing, implement innovations, and gather empirical data. By treating industries as laboratories, researchers can explore and validate theories, methodologies, and technologies in authentic working environments, which often leads to more robust and applicable outcomes.

This paper is articulated as follows: in Section 2 we will discuss our relationship with ASML and the background knowledge needed to better understand the foundations of the refactoring work. In Section 3 we present related work in the domain of software maintenance. In Section 4 we will describe the Action Research methodology used for the refactoring task as ASML. In Sections 5 and 6 we introduce the Cleaning-Up Cycle and analyze its performance. To conclude, in sections 7 and 8 we provide a discussion and a conclusion.

II. Background

A. ASML

Advanced Semiconductor Materials Lithography (ASML) is a leading Dutch manufacturer of lithography machines, specializing in cutting-edge solutions for semiconductor manufacturing. Known for its state-of-the-art technologies, ASML enables the production of microchips with enhanced efficiency, smaller dimensions, and improved processing capabilities.

Since its establishment in 1984, ASML, like many technology companies, has been developing and maintaining internal software. Despite ongoing improvements due to changing requirements and discovered bugs, much of the established code remains unchanged. The recent refactoring project focused on the ”reticle align” repository, which is part of a larger functional cluster responsible for various image-related tasks.

Over more than 20 years of development, the system has evolved, accumulating architectural technical debt. Initially, it consisted of a single monolithic component closely tied to another common component. This monolithic component handled most operations within its layer, following ASML's vertical slicing, layered architecture. Due to its size and responsibilities, the team extracted a more specialized subcomponent for specific image-aligning tasks and parallel processing. However, due to tight deadlines, the refactoring process didn't strictly adhere to proper architectural and component-level design patterns.

This led to the introduction of multiple component-level dependencies and hundreds of illegal symbol-level dependencies. Despite following established architectural practices to decouple large components into smaller ones, the new component introduced unwanted dependencies, violating the principles of the layered architecture [21].

Considering the size of the functional cluster and limited resources for maintenance, implementing SOLID principles required careful planning. The refactoring work aimed to realign the components to the original intended architecture.

B. Relationship with ASML

The academic/industry relationship that exists between the University of ABC and ASML was established in the last 10 years. Besides obvious disciplines such as Physics and Mechanical Engineering, ASML also considers Artificial Intelligence, Computer Science, and Mathematics to be “critical competencies” for their business. As such, a strong relationship exists between the department of ABC: over 20 BSc students have so far worked on ASML-based software development projects, and some 15 students have engaged with ASML-based refactoring and reengineering projects.

This experience report was conducted by one BSc student as part of their Honours Program: this program offers highly motivated students of the University of ABC the opportunity to pursue interdisciplinary coursework, engage in research projects, and participate in extracurricular activities to enhance their academic experience. The student engaged earlier with ASML, as part of a software development course, having experience with multiple parts of the ASML codebase and their necessities for almost a year, before starting the refactoring work outlined in this paper.

C. The Q Interface

We are dedicating this subsection to a better understanding of the ”*Q interface*” as the core refactoring objective of our work. In our context, an interface is a concrete C++ component that contains the functionality described below.

The Q interface was written so that strongly typed code could communicate with the ASML system. The ASML system contains a component that can process virtual typed data (such as Python data structures that do not have an intrinsic type associated with them). If a client develops code in a strongly typed language (e.g. C++), they cannot interact with this component from the ASML system. To allow that, the Q interface was introduced to bridge the strongly typed code and the ASML system. We call developers who would write code for accessing the ASML system for this data processing ability, clients.

As guidelines evolved within the ASML ecosystem, it was decided that clients must be prohibited from writing strongly typed code altogether. With this new rule enforced, the Q interface became deprecated as it was no longer used.

Moreover, some of the Q interface functionality was also utilized internally by the system itself. Therefore, a simple removal of the interface was not an option. In figure 1 we see the desired outcome of our refactoring. We implemented the Q interface functionality internally and removed the interface.

If the interface is left untouched, the main issue is that the Interface Segregation Principle (ISP) could be violated if clients decide to create dependencies to this deprecated interface, as their code should not depend on functionality they do not need. Furthermore, it adds unnecessary complexity to ASML's build infrastructure. All interface objects must be built in a dedicated way such that they are seen externally

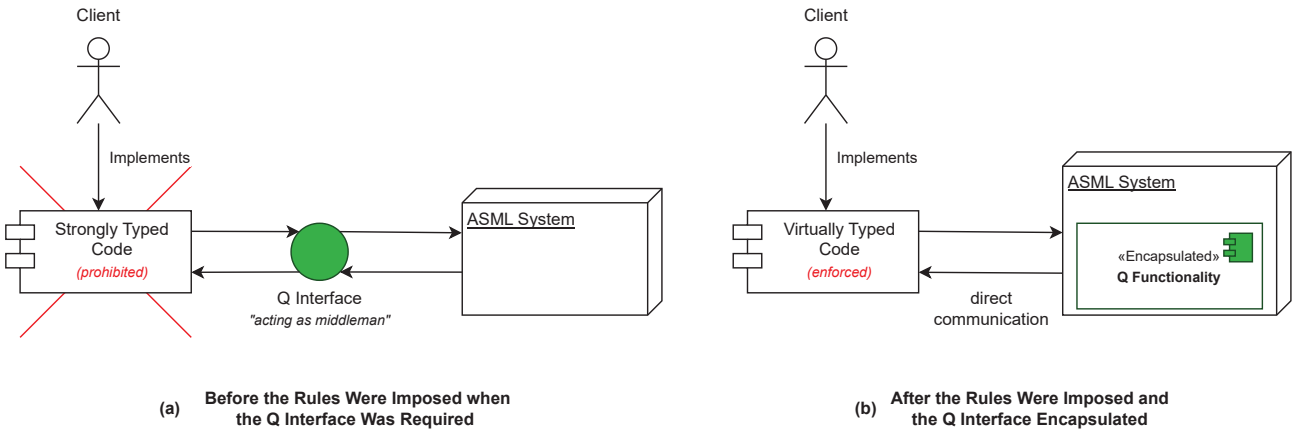

Fig. 1: Performed refactoring context: *(a)* initial state before the rules regarding strongly typed implementations were imposed, *(b)* refactoring outcome for one interface instance after rules were imposed

by all the other components. Removing the interfaces would simplify the codebase considerably. Therefore, our work can be defined as a preventive refactoring that also reduces the project's complexity.

The total number of Q interfaces is currently too large to be completed in the timespan of our refactoring work. In addition, the nature of the task is quite repetitive. Considering these two factors, we aimed to devise a refactoring strategy that could be passed on to future employees. Using the Action Research methodology, we constructed the **Cleaning-Up Cycle**, a strategy that we refined during our work and that we put into practice. The Cleaning-Up Cycle is an approach that increases the time efficiency of refactoring and minimizes project dependency errors. It is aimed at ISP-related issues, especially in highly coupled systems.

D. Data Availability

The *Cleaning-Up Cycle* strategy was developed while conducting refactoring work on ASML's system. We were under NDA agreement during the entire process as the codebase is highly confidential. We provide all the information as normalized percentages (time efficiency, number of interfaces) to guarantee the confidentiality of data. For these reasons, we are unable to give access to the code implementation.

We provide the following information for replication to the best of our ability. We operated on a C++ codebase organized by different nested folders and built using multiple "makefiles". The Q interfaces were part of a different folder branch than what we call the "ASML System" in Fig. 1. The C++ Q interfaces had to be adapted for an internal implementation in the system by modifying the passed parameters and how they were accessed. To remove the interfaces, multiple makefiles had to be updated. Furthermore, the unit tests associated with said interfaces had to now point towards the internal implementation. We consider our method to be applicable outside the ASML environment granted to this description that depicts the general contents. Our method is most suited for the objective of extracting functionality while removing the original class/interface of said functionality. Another crucial aspect is its optimization for highly coupled systems. For systems in which this does not represent an issue, the Cleaning-Up Cycle might not provide as high of a benefit.

III. Related Work

The experience report [22] notes the distinction between small-scale and large-scale refactoring. While the former is executed at the same time as development, the latter is represented by a large dedicated timeframe only for refactoring. In [23] it is concluded that large-scale refactoring can improve the maintainability of software, whereas the outcome of a single small-scale refactoring is hard to predict. The refactoring of Q interfaces is considered a large-scale refactoring.

A literature review analyzing over thirty years of work [24] groups more than 3,000 papers into different categories. We define our paper as a recommendation study [25], [26], [27], [28] on performing code cleaning and modularization in a highly coupled legacy system. We focus on an architectural artifact [29], [30], [31] by encapsulating the Q interfaces under the object-oriented paradigm. We improve both the internal [32], [33], [34] and external quality [35], [36], [37], the latter representing the main objective. We carried out an industrial evaluation [38], [39], [40], [41] within ASML.

Refactoring can be categorized by the level of automation into three types: automatic, semi-automatic, and manual. For instance, Moij et al. [42] emphasize a "code aligning phase" before an automatic model-based rejuvenation to enhance programmer-centric code style and uniformity. Griffith et al. [43] explored automated genetic algorithm-based approaches. Both techniques could not be applied to our case due to different objectives. Our study primarily aligns with the "Extract Method" type of automatic refactoring [44], a method that may exhibit reluctance to undertake, especially when relying on IDE support. In our case, no tools were available that could perform the automatic extract method.

Additionally, several studies in the industry have utilized a model-based semi-automatic approach. Dams et al. [19] aim to

enhance code uniformity and streamline project dependencies. Despite the insights provided by the study, we encountered challenges in directly translating their results to our cleanup task.

The largest body of work we could consult was based on manual refactoring. In past studies it was preferred as noted by Schuts et al. [45]. Manual empirical studies, such as those by Park et al. [46], Nguyen et al. [47], and Sousa et al. [48], analyzed software quality standards, and highlighted the preference for manual investigation over automated tools. Furthermore, automatic refactoring is usually preceded by manual work [49], [50].

Another interface clean-up is presented in [19]. Their approach allows for semi-automation since their primary goal is to improve code uniformity and simplify project dependencies. We opted for manual refactoring because while the general refactoring strategy might stay the same, each interface requires different implementation details. We could not define transformations for our work. Schuts et al. [45] use active learning as a semi-automatic technique. This is due to the lack of formal specifications when replacing an old component. The ASML system provides well-defined specifications due to the reliability needed by the lithography machine.

Manual empirical studies [46], [47], [48] analyze software quality standards, such as code clones and code smells. More studies manually investigate the performance of semi-automatic or automatic tools [51], [52], [53]. We observe that manual investigation is desired to verify the correctness of assisting tools which focus on the implementation phase.

We decided against an automatic refactoring tool because of the complexity that it would have required. To the best of our knowledge, no existing tool would have fitted our project description, and creating one of our own would not outweigh the cost.

The industry-as-lab approach involves leveraging real-world industry settings as experimental environments for research and development, facilitating practical testing, and implementing innovations. The previously mentioned work by Dams et al. [19] on refactoring uses the industry-as-lab approach. Another industrial case study is performed at Xerox which investigates the documentation of refactoring [20]. They concluded by stating the importance of following a procedure when documenting refactoring activities. Our work focuses on that, by delivering a standardized routine to be adhered to by developers when undertaking our task.

In a nutshell, after identifying the different categories in which our work is placed, we define the experience report of this paper as industry-as-laboratory.

IV. Methodology

This paper follows the ACM empirical guidelines suggested to perform Action Research[2]: in particular we have considered the 13 'Essential Attributes' contained in that empirical standard (e.g., (i) *justifies the selection of the site(s) that was(were) studied*; (ii) *describes the site(s) in rich detail*; (iii) *describes the relationship between the researcher and the host organization*; etc.) to describe the methodology, results and lessons learned from this experience.

[2]https://www2.sigsoft.org/EmpiricalStandards/docs/standards?standard=ActionResearch#

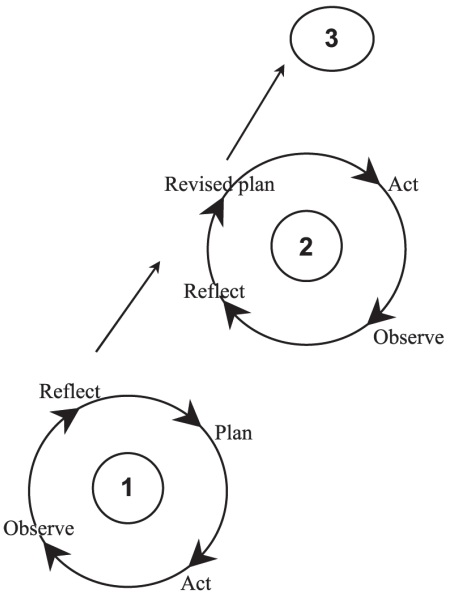

Fig. 2: The Action Research Spiral from [55]. Our applied AR methodology for proposing a refactoring strategy follows the depicted steps.

At its core Action Research (AR) follows three axioms [54]:

1) action research is about people reflecting upon and improving their own practice
2) by tightly inter-linking their reflection and action; and
3) making their experiences public to other people concerned by and interested in the respective practice.

In our research, an AR cycle would span the refactoring of one Q interface. After six Q interface refactorings, our **Clean-Up Cycle** underwent six AR spirals. We provide the final product of this iterative process. We discuss concretely the steps of AR in the following subsections.

A. Planning with SOLID Principles

The AR spirals are depicted in Fig. 2. We start the AR by planning the goals of the refactoring process and the procedure it must follow to achieve them. We then act according to the plan and by observing and reflecting, we can revise the plan. We repeatedly perform this procedure until we arrive at a sensible outcome. In our case, we produced the refactoring strategy, the Cleaning-Up Cycle.

For the planning phase, we established how the refactoring should be performed. We saw that most of the Q interfaces could be divided into three distinct categories, classified by the expected complexity of the refactoring. More information will be presented on this in VI-A.

Then, we proposed the implementation of the refactoring and analyzed if the violation of the ISP was avoided. The ISP states: *"Clients should not be forced to depend on interfaces that they do not use"* [17]. To achieve this, we planned the

removal of the Q interface and its functionally re-implemented internally, such that clients will not be able to implement an interface they must not use. We note that the other 4 SOLID principles must also be respected after the refactoring guaranteeing that no new technical debt is introduced. During planning, we also observed how the components affected by the refactoring were declared in the build system configuration and how the corresponding unit tests were written. Establishing how to address issues raised by these factors was part of the plan. We present how we approached them in the next subsection.

To end the planning phase, we notice that while all Q interfaces are fundamentally similar, they present small variations in their implementation and the context in which they can be found. This means that we need to allow for flexibility during acting.

B. Acting & Observing

Following the planning phase, the AR we performed is mainly categorized by performing six rounds of refactorings on the Q interfaces. The refactoring itself is equivalent to an *acting* phase in Fig. 2. Afterward, during *observing* we compared previous installments with the most recent one. This included different metrics such as time spent (as a percentage of the total time spent on all the interfaces we refactored) and the number of failures. We call a "failure" an instance in which we had to rewind to a previous working version of the project due to introducing too many dependency errors. This is a standard software development procedure facilitated by versioning tools [56]. We chose this as a metric since we consider it a direct consequence of operating on a highly coupled system.

During the initial iterations of acting, we did not have a recommended structured approach to being with. We would start by translating the implementation internally, deleting the interface, and attempting to fix the dependencies. This initial separation of concerns led to many issues. We observed that resolving the dependency problems was particularly difficult. As our strategy matured, we observed that we incorporated intermittent phases of dependency updates. Another change we observed to fluctuate across the six AR spirals is the timing at which we performed our testing. Testing the ASML system needs to be scheduled as a job and can impact the efficiency considerably. At first, we were performing it too sparsely leading to hard to uncover errors. Then, the opposite happened and a lot of time was spent on testing correct code configurations. By the end, we identify three ideal moments when testing should be conducted. The entire process is described in Sec. VI which refers to our final result from AR.

In order to improve our acting, we used our observations in the reflection phase.

C. Reflecting

After we successfully finalized the refactoring of one Q interface, we reflected on the respective iteration in order to revise our plan. The build system configuration represented one of the largest time killers, due to the high number of dependencies between components. The long test times and building times also represented a bottleneck, as a lot of waiting had to be done to guarantee correctness. This led us to see the high stakes of writing correct code.

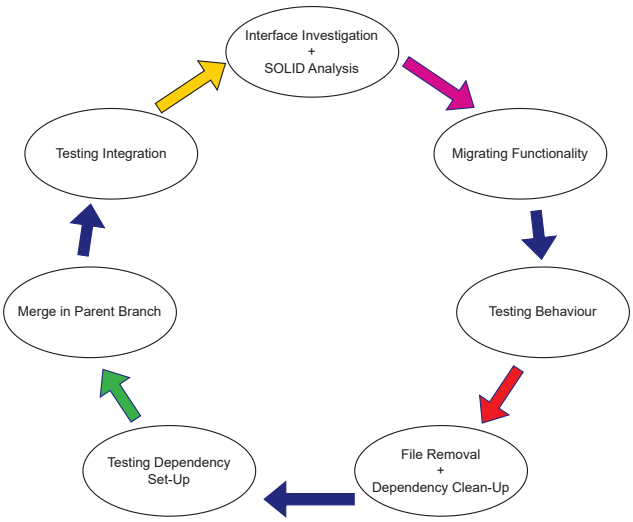

Fig. 3: Clean-Up Cycle Used on Interface Encapsulation.

We had to isolate the different tasks that we would perform in such a way that the project could be compiled and tested immediately. As part of our work, we also had to refactor the unit tests themselves and during the AR spirals, we had multiple attempts at separating the refactoring of the unit tests from the concrete project implementation. This led to even more issues with testing the project and we decided on performing both simultaneously. Another strategy we developed with AR was to allow for an intermediate phase in which the code is duplicated. This way we could test that the internal translation was successful without worrying about breaking the overall structure.

With each interface attempt, we followed the described steps and revised our plan to improve on the listed problems. After iteration five of AR, our process stabilized, and we did not change it in iteration six.

V. The Cleaning-Up Cycle

After iterating through the presented AR framework, we identified the **Cleaning-Up Cycle** (Fig. 3). The main steps of the cycle are enumerated below:

1) **Interface investigation** - For the first step, we analyze the Q interface that needs encapsulating. We are looking at all the places in which the interface was referenced, and the type of functionality it provided, and we proposed a solution that would satisfy the refactoring. We accumulate information on what dependencies are expected to break and confirm that our proposed solution is feasible. Some of the encountered issues while approaching different Q interfaces were related to where should the functionality of the interface be encapsulated and to adapt the unit tests. To arrive at a consensus, discussions with the functional architect of

ASML were ongoing and we settled on an approved approach. We used the SOLID principles to verify the quality of work.

2) **Migrating Functionality** - In the second step, we perform the encapsulation of the Q interface functionality internally. We do not remove any original files or methods, however, we make all the necessary modifications s.t. we obtain an equivalent system that executes no calls to the original interface. This step acts as a checkpoint in our process and guarantees that all the required components have been internalized successfully. Any errors related to the migration would be identified and solved early before any issues related to the dependency removal could be introduced.
3) **Testing Behavior** - As mentioned, testing the system requires scheduling a job. The execution of the testing job can take up to thirty minutes, depending on how many components are in its scope. To balance the number of performed testing jobs, we found that testing the behavior once after the internal implementation is adequate.
4) **File Removal** - The actual clean-up is performed and any unnecessary files are removed. In the "File Removal + Dependency Clean-Up" step, we search for any references to the original files, update or delete them as required, and delete all the files related to the Q interface. Modifications to the system build routines were critical in this step, as the highly coupled layer architecture of the codebase would reference targets to the Q interface in numerous places. If these targets were not correctly modified or removed, it could lead to errors that were difficult to trace back.
5) **Testing Dependency Set-Up** - Since many changes to how the dependencies are set-up in the build routines have been made in the previous step, a second testing phase is required now. These dependencies affect both unit tests and the source code, therefore unforeseen issues can be identified at this point.
6) **Merge in Parent Branch** - The changes corresponding to the interface refactoring are merged into the parent branch. The parent branch contained all the Q interfaces that we have refactored until that point, and it would represent our final delivery for ASML.
7) **Testing Integration** - For the final phase, a system-wide testing job is performed to guarantee that all the components behave correctly together given that they often depend on each other. This job can take up to a few hours.

Summarizing the clean-up cycle, it represents a multiple-phase process in which each testing phase acts as a checkpoint in the refactoring. The process was developed to minimize the number of errors introduced, therefore making developers more time-efficient when approaching this task.

It is a product of the AR methodology and can be referred to as the "treatment" in its context. It evolved during our research iterations. Among others, the framework helped us to decide on the most opportune moments for scheduling testing jobs and to separate the concerns of our refactoring in a way that allows for intermittent compilation. Initially, we would attempt to clean up dependencies gradually which paradoxically led to introducing difficult to uncover errors. We found out that by separating the migration phase and the clean-up phase, we were able to discover the origin of issues faster.

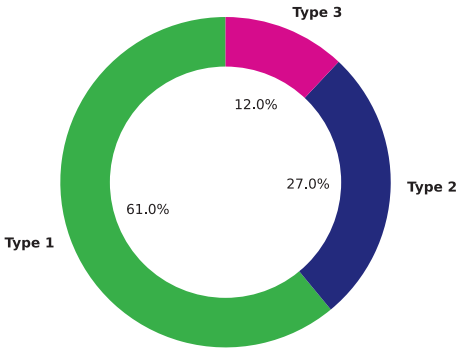

Fig. 4: The Distribution of all the Interface Clean-Up Types

VI. Cycle Performance

In this section, we will evaluate the effectiveness of the proposed cycle by examining the performance of the Q interfaces that have been completed. A total of six interfaces were refactored, and categorized into three distinct types using the open card sorting technique [57]. We were given access to the entire software that included each one of the interfaces and we identified common characteristics that were relevant to the refactoring. We found that we could group the interfaces by the expected complexity of the associated refactoring.

To achieve anonymity and respect the Non-Disclosure Agreement with ASML, we have excluded below all the absolute values when referring to any information related to ASML's codebase. We have instead provided our findings as normalized percentages. We measured the distribution of the different types of refactoring for total number of Q interfaces that are required to undergo refactoring, time spent per finalized individual Q interface (presented as a percentage of the total time spent on all the finalized interfaces), and the number of attempts per finalized Q interface (provided as an absolute value as it does not disclose any information).

A. Refactoring Types

We categorized the types based on the anticipated difficulty of refactoring, with Type 1 being the easiest and Type 3 the most challenging. To assign each interface to a category, we conducted an initial analysis of the entire domain, including those not directly modified. As a result, we identified specific attributes for each type. The distribution of types can be seen in Fig. 4. The attributes defining each type are as follows:

- Type 1: The Q interface is unused and can be safely deleted without local encapsulation.
- Type 2: The Q interface has a single responsibility and is utilized in only one location within the module. Interfaces in this category are typically addressed systematically.
- Type 3: The remaining interfaces fall into this category. They either have multiple responsibilities requiring internal encapsulation or are utilized in multiple locations. These interfaces pose a greater challenge, requiring developers to determine specific solutions.

From Fig. 4, we observe that the majority of interfaces belong to either the first or second types. This indicates that most of the technical debt introduced by the Q interfaces could be addressed relatively easily. Furthermore, Type 2 imposes specific constraints on dependencies and functionality, suggesting that a systematic approach could benefit the entire class. Although cleaning up Type 3 interfaces may require additional creativity, our cycle still offers a promising foundation.

By dividing the refactoring process into multiple phases, developers are equipped with a divide-and-conquer mindset from the outset when tackling the problem.

B. Comparative Studies

As stated in Section III, our work can be summarized as a manual extract method refactoring. The extended literature review on this type of refactoring by AlOmar et al. [44] analyzes 83 similar studies. We performed our refactoring to prevent the violation of the ISP. Among the 86 studies, AlOmar identifies three other driving factors that do not compare with ours: code clones, long methods, and the separation of concerns.

Additionally, we find that *"most of the studies used a dataset created by the paper's authors, corresponding to 86.74%"* and that *"experiment-based validation is the most widely used method, with 59.03% of the studies that use it"* [44], which leaves us with the inability to perform an external comparison. We decide to perform an internal experiment-based validation by comparing the efficiency of the refactoring with and without a fully matured cycle. To ensure consistency and eliminate any discrepancies due to varying skill levels among developers, all field refactoring tasks were exclusively carried out by the primary author of this paper.

C. The Cycle in Practice

We successfully refactored a total of six Q interfaces before the end of our contract. In the end, one single delivery containing the modifications of all the interfaces was provided to our supervisor for inspection. Out of the six Q interfaces, we encountered two of each type.

In this section, we will present how our clean-up cycle evolved and affected our performance over time. All the refactoring was performed by the main author of this paper in order to avoid any variance from skill discrepancy across multiple developers. We will execute our analysis by looking at both the chronological order in which we completed the respective interfaces and the number of failures, while also analyzing the impact of the interface type. The number of failures is measured by counting how many branches were created for each interface (as one branch corresponds to one attempt) and the time spent is observed by looking at the used versioning tool.

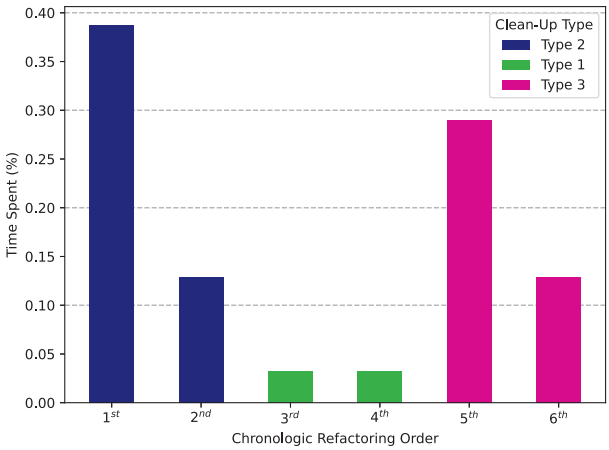

Fig. 5: Percentage of Time Spent on Interface Against the Order in Which They Were Executed.

In Fig. 5 we observe the order in which we approached the refactoring. We first completed the Type 2 interfaces, then the Type 1 interfaces, followed by the last Type 3 interfaces. Initially, the order was arbitrarily chosen, as we formulated the Q interface type definition only later. We measure the fraction of time we spent on each interface against the order of execution. As expected, the first Q interface took the longest time. This can be due to the initial unfamiliarity with the domain and the lack of a structured approach. The clean-up cycle was formulated during an iterative process, therefore in the beginning there was nothing. After completing our first interface, we reflected on our performance under the AR methodology. Since the 2^{nd} interface was from within the same type as the first one, and the plan was revised once, it was completed much faster.

The Type 1 interfaces fall into the easiest-to-refactor category. They provide no dependencies and their functionality does not need to be encapsulated. The phases in our clean-up cycle do not apply to them. In the following analysis, we will provide them to show a complete picture, but they should be taken as outliers. The 3^{rd} and 4^{th} interfaces took the least time due to their simplicity.

A surprising result is reached when advancing to the Type 3 interfaces. Their added complexity resulted in significantly more challenging investigation and implementation phases. In these cases, we had to run our proposed refactoring plan through the SOLID analysis more thoroughly. The Type 3 Q interfaces we encountered presented multiple functions that had to be encapsulated and more dependencies than the rest. We also encountered unprecedented system build errors. Despite these added challenges, we still observe a decrease in the percentage of time spent when comparing them to the Type 2 interfaces. By the time we reached this stage, we already had a well-structured Clean-Up Cycle that allowed us to isolate

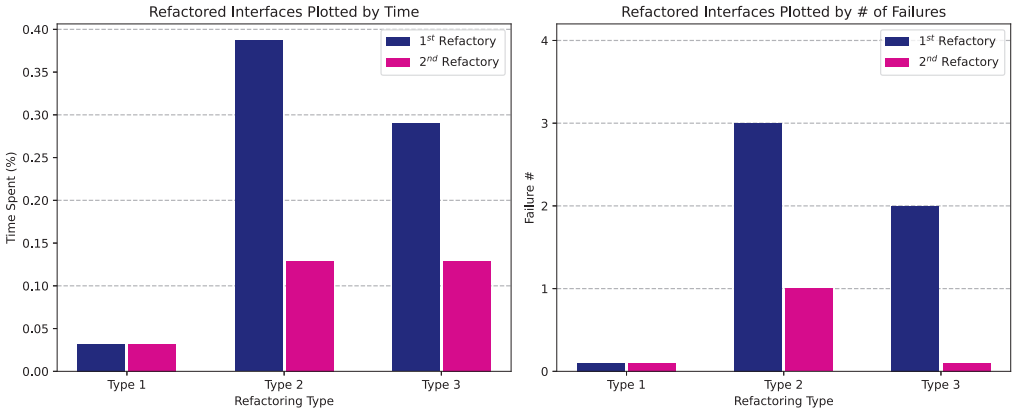

Fig. 6: Correlations Between Effort Spent on an Interface and its Type: *(left)* Type Plotted Against the Percentage of Time Spent, *(right)* Type Plotted Against the Number of Failures when Refactoring

the phases of the refactoring. Even though we encountered new issues, we were able to identify their root cause quickly owning to the modularity of the cycle.

In Fig. 6 we elaborate further on our experience. This time we group our work on the Q interfaces by their types and present one additional metric (on the right), the number of failures until the interface was successfully refactored. We define a failure as a moment in which we have to start over the process due to untraceable errors. In each group, we see the first and second interfaces we attempted of each type.

As seen above, Type 1 provided no difficulties. We also observe a clear trend where the second interface in each group benefits from reduced time spent and fewer failures. In Types 2 and 3, the second interface had one failed attempt for Type 2, while Type 3 had none, which we attribute to the cycle's maturity by that point.

As a final remark, the Type 2 and Type 3 interfaces differed in implementation details and task complexity. We attribute the drop in time spent and failures to the maturity of our cycle.

D. Statistical Analysis

To validate our results we perform an unpaired double-tailed *t* test. We chose the *t* test over the *z* test due to the continuity of the time data. As explained above, we do not include the Type 1 interface in our statistics since we consider them outliers due to their simplicity. Looking at types 2 and 3, we classify the first approached interface of each as *treatment I* and the second interface as *treatment II*. This data is shown in Fig. 7 alongside the *mean difference* bar, aiding in the visualization of the test significance.

Executing the unpaired double-tailed *t* test we obtain the *P*-value equal to 0.0496 ($P = 0.0496$). With the conventional significance level ($\alpha = 0.05$) we conclude that the proportion of time spent is statistically different and that the treatment of applying the matured Cleaning-Up Cycle significantly decreases the time spent per refactoring. We note that the P-value is very close to the 0.05 significance level. This value could be further improved by performing a one-tailed test instead or by obtaining more data points.

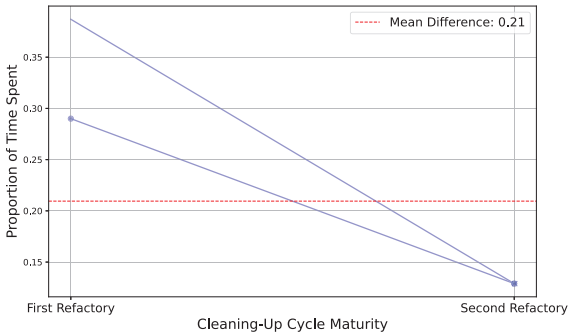

Fig. 7: Statistical Visualization of How the Proportion of Time Spent per Interface Evolved with the Maturity of the Cleaning-Up Cycle.

E. Codebase Outcome

The final results are represented accurately by Fig. 1. Furthermore, relating to the subsection II-D on Data Availability we acknowledge that the six Q interfaces were removed. The functionality that was still required by the ASML system was implemented internally. This was done by migrating the associated methods and changing their signatures. After this, multiple nested makefiles responsible for building the systems had to be modified. Moreover, because we migrated functionality while also changing function signatures, many associated unit tests were updated to reference the right re-implemented method. This guaranteed that no test coverage was lost as a result of our work. In the end, the complexity of the system was reduced by removing the dependencies on the Q interfaces, and the violation of the ISP was avoided.

a) Internal Review: To verify the correctness and value of our refactoring pattern we reached back to ASML after we finished our project. The lead software designer of the team mentioned that: *"We are grateful for the diligent work of XYZ in the refinement of our software facilities' legacy external interfaces. XYZ's application of their academic in-*

sights has played a pivotal role in enhancing the overall client experience. By systematically cleaning up these interfaces, XYZ has demonstrated a keen understanding of both theoretical concepts and practical implementation, resulting in tangible improvements for our users. We commend XYZ for their dedication and contribution to optimizing our software infrastructure, which undoubtedly strengthens our commitment to delivering exceptional service to our clientele." This statement is used to validate the implementation itself. By client experience, it is referenced the decrease in complexity that the encapsulation process brought. When asked about the provided documentation and clean-up cycle strategy, it was confirmed that they will serve as the foundation for future employees embarking on the same task.

VII. Discussion

The goal of our work within ASML extended past simply performing system maintenance. We aimed to pass down our experience with the project to future employees. For that, we used the AR methodology to discover a tailored strategy that can be replicated.

We believe that the Clean-Up Cycle is not domain-specific and can be applied outside the ASML ecosystem. Our approach reduces the required testing phases while allowing the project to be intermittently compiled. Looking at the cycle, we see three testing phases that follow a refactoring phase. They are positioned to separate the concerns, each testing phase acting as a checkpoint in the refactoring process. The cycle forces the maintainer to discover errors early and in a small enough context to identify the cause.

While we conducted our work on the Q interfaces, we encountered difficulties that might be system-specific. We share our experiences with how we overcame them. We avoided introducing new bugs by updating the existing unit tests during the "file removal + dependency clean-up". Due to their continuous operation with extremely high-reliability requirements, all software surrounding the lithography system is thoroughly tested. During code cleaning, no tests were removed, and all were updated at the same time with the migration. As no new functionality was added, no test coverage was lost and no new bugs were introduced. At first, we were not familiar with the system's complexity. The AR methodology allowed us to slowly build up our understanding. As we gained more domain knowledge by applying the planning, acting, observing, and reflecting phases, we developed the general strategy presented in this paper. To guarantee alignment, we underwent discussions with the supervisor from the host organization. Lastly, the refactoring of Q interfaces was deemed costly from a time perspective. Multiple future developers will be tasked with it. To reduce the total time spent, we were initially asked to develop the aforementioned strategy.

A. Implications for Researchers

Our experience report provides researchers with a framework that combines the action research methodology with the industry-as-laboratory technique. Following the AR steps in an industry setting allows for the optimization or testing of software engineering practices. Such practices are not limited to improving the time efficiency of the refactoring process, but can also aim at improving metrics such as code quality, usability, or refactorings like behavior-preserving transformations, microservices migration, automatic refactoring tools, etc. We leave research that does not focus on manual refactoring for future work.

We also encourage researchers to use existing versioning tools when operating in industry-as-lab settings. In our experience, the ASML environment facilitated the experiment setup, thus we encourage the collaboration between academia and industry, as we show by example how both parties can benefit.

Finally, we found that legacy systems can pose their own challenges. Sometimes, in such instances, the software can not be rebuilt from scratch, either due to its complexity or high migration costs. We identify the banking sector [58], aerospace [59], [60], healthcare [61], [62] to be dealing with such issues and not only [63]. The ASML lithography system is one such instance. Researchers employed by companies or institutions could profit by providing modernization solutions. With the rise of LLM solutions, they could aid in this task by identifying and prioritizing weaknesses in legacy code.

B. Implications for Practitioners

The Cleaning-Up Cycle represents a good strategy for developers or new hires tasked with code maintenance who are not familiar with the domain of the system. It is a structured approach that increases the time efficiency and reduces the number of failed attempts (code-breaking bugs that are hard to trace back).

From the lessons we learned working on ASML's software, we recommend that developers who work on machine software that can not be rebuilt from scratch pay extra attention to its future proof. As the software gained in complexity across decades the build time and testing time became considerably large. We had to adapt our refactoring strategy according to these weaknesses. We emphasize that scalability can dictate the longevity of many engineering feats. Such products include and are not limited to autonomous vehicles, biotechnology systems, space telescopes, prosthetics, and automated assembly lines. This is further underscored by the fact that software is often reused for products from within the same category, as lithography software was reused in many product lines. We also encourage that principles, such as SOLID, DRY [64], KISS [65], be enforced from the earliest stages of development. Software aging [13] is not considered a new concept anymore, and standardizing the software early according to the desired principles can reduce maintenance time.

We also observed that changing versioning tools becomes difficult for projects of this size. A migration for a system like the one on which we operated can span years, therefore we draw attention to the importance of DevOps tools, especially for complex mechanical/engineering systems that might overlook this aspect.

C. Limitations

Our study has several limitations: first, it explores a relatively narrow range of refactoring issues and technical debt challenges in legacy systems. Even so, showing how these play a role in one of the largest Dutch industries can give an insight into other industries facing similar challenges.

Second, our refactoring cycle is based on a mostly manual approach, as we did not investigate the possibility of incorporating semi-automatic or automatic refactoring tools. Once the refactoring strategies become clearer and more central to the maintenance effort, such semi-automated or fully-automated approaches will become more standard.

Third, the replication and application of the Cleaning-Up Cycle in other contexts can be time-consuming, especially in the Initial Assessment and Planning phase. The identification of problematic areas of the legacy code, such as those with deprecated interfaces or high technical debt, requires access to developers with experience in the existing codebase, as well as to developers with expertise in refactoring and software design principles.

Fourth and final, a long-term impact assessment is needed to evaluate the sustained impact of the Cleaning-Up Cycle on code quality and maintenance efforts. We have gathered initial feedback on the effectiveness of the refactoring Cycle, but only a longer time observation can show whether this refactoring strategy should be adapted and evolved based on new challenges, technological advancements, and changes in project requirements.

D. Threats to Validity

As with any other experience report, an action research paper faces several threats to validity. We highlight below a number of those, and how we limited their effect on our study.

The main threat to internal validity is that the refactoring process might have become faster and more efficient due to the technical knowledge gained during the process itself.

By working on different types of interfaces (especially Type 2 and 3), the technical knowledge we obtained is not as transferable as it would be if we were only working on the same type. We measured two metrics, ‘time spent’ and ‘number of failures’. After developing our clean-up cycle, we observed better performance on both even though we moved to the domain of Type 3 interfaces, which were more complex. The switch from Type 2 to Type 3 nullifies the effect of our better understanding of the codebase. This effect should be further investigated by future employees taking on the same task by analyzing their initial performance when following our strategy.

The findings may suffer from external validity since we worked in the context of ASML’s legacy codebase and might not be generalizable to other industries or organizations with different software development practices. To prevent this we abstracted specific technical details. We present a strategy for refactoring interfaces that require encapsulation or extraction. The context of our problem is a highly coupled system known not to be unique.

Construct validity may be threatened by the number of interfaces we completed the refactoring for. Excluding Type 1 (which could be considered outliers due to their simplicity), we provide four interfaces, two of each remaining type. From our measurements, we can conclude the effectiveness of the clean-up cycle, granted the discrepancy in complexity and the order in which we executed them. More data points could both be detrimental and beneficial for this study. With more data points the internal validity issue of obtaining domain knowledge could only increase. This is also due to lacking additional different types. More data points could be beneficial by further improving the confirmability of our study.

VIII. Conclusion

This paper described an experience report of a refactoring work on legacy code, performed at ASML, the leading Dutch manufacturer of photolithography equipment, at the core of the semiconductor manufacturing industry. While adapting new software standards, many ASML interfaces became obsolete, and their presence in the codebase represented a potential risk for violating the Interface Segregation Principle (ISP).

Our refactoring approach was carried out using an Action Research template: the *planning* for refactoring was guided by (i) the types of interfaces present in the code and (2) their adherence to the SOLID architectural principles. The *action* phase (e.g., the actual refactoring) was carried out in a way that could be reproducible and extended in the future to the other types of interfaces, and by the ASML engineers when taking on a similar task. For the *observation* phase, we took note of the efficiency of the refactoring process, in terms of the time taken to act on each type of interface and the number of failed attempts per interface. Finally, the *reflection* phases assessed the outcomes of the refactoring process. During this phase, we developed the clean-up cycle to improve subsequent work. After the refactoring work, the interfaces were correctly encapsulated, and the codebase became more maintainable, by applying SOLID principles.

As the two contributions of this paper, we have shown the applicability of one SOLID architectural principle (i.e., the ISP) in a real working environment, thus contributing to the industry-as-lab research approach. We have also contributed a replicable clean-up refactoring cycle, that can be used as a general extraction and encapsulating strategy in the refactoring of highly coupled systems.

REFERENCES

[1] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 73–87.

[2] C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, B. Penzenstadler, N. Seyff, and C. C. Venters, "Sustainability design and software: The karlskrona manifesto," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 467–476.

[3] J. Ransom, I. Somerville, and I. Warren, "A method for assessing legacy systems for evolution," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. IEEE, 1998, pp. 128–134.

[4] K. H. Bennett, M. Ramage, and M. Munro, "Decision model for legacy systems," *IEE Proceedings-Software*, vol. 146, no. 3, pp. 153–159, 1999.

[5] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative reengineering of legacy systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 225–241, 2003.

[6] S. Matthiesen and P. Bjørn, "Why replacing legacy systems is so hard in global software development: An information infrastructure perspective," in *Proceedings of the 18th ACM conference on computer supported cooperative work & social computing*, 2015, pp. 876–890.

[7] S. C. Dunn and M. H. Sawyer, "A proposed approach for prioritizing maintenance at nasa centers," in *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, 2013, p. 108.

[8] A. J. Koski and T. J. Mikkonen, "Taming a monster: Tackling the emergent issues encountered in mission critical system development," in *International Conference on Agile Software Development (XP 2017)*. Agile Alliance, 2017.

[9] U. Knop, P. Hofman, M. Mihatsch, and M. Siegmund, "Balancing variability and costs in software product lines: An experience report in safety-critical systems," in *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume A*, 2023, pp. 213–222.

[10] C. Francalanci and F. Merlo, "The impact of complexity on software design quality and costs: An exploratory empirical analysis of open source applications," 2008.

[11] E. Capra, C. Francalanci, and F. Merlo, "The economics of community open source software projects: an empirical analysis of maintenance effort," *Advances in Software Engineering*, vol. 2010, 2010.

[12] S. Dashevskyi, A. D. Brucker, and F. Massacci, "On the security cost of using a free and open source component in a proprietary product," in *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings 8*. Springer, 2016, pp. 190–206.

[13] D. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, 1994, pp. 279–287.

[14] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[15] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?" *Journal of Systems and Software*, vol. 169, p. 110710, 2020.

[16] R. Verdecchia, P. Kruchten, P. Lago, and I. Malavolta, "Building and evaluating a theory of architectural technical debt in software-intensive systems," *Journal of Systems and Software*, vol. 176, p. 110925, 2021.

[17] H. Singh and S. I. Hassan, "Effect of solid design principles on quality of software: An empirical assessment," *International Journal of Scientific & Engineering Research*, vol. 6, no. 4, pp. 1321–1324, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:212553166

[18] C. Potts, "Software-engineering research revisited," *IEEE Software*, vol. 10, no. 5, pp. 19–28, 1993.

[19] D. Dams, A. Mooij, P. Kramer, A. Rădulescu, and J. Vaňhara, "Model-based software restructuring: Lessons from cleaning up com interfaces in industrial legacy code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 552–556.

[20] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 348–357.

[21] A. Sharma, M. Kumar, and S. Agarwal, "A complete survey on software architectural styles and patterns," *Procedia Computer Science*, vol. 70, pp. 16–28, 2015, proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems.

[22] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini, "Industry experiences with large-scale refactoring," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov 2022, pp. 1544–1554.

[23] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Empirical study on refactoring large-scale industrial systems and its effects on maintainability," *Journal of Systems and Software*, vol. 129, pp. 107–126, 2017.

[24] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig, "30 years of software refactoring research:a systematic literature review," 2020.

[25] I. Verebi, "A model-based approach to software refactoring," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 606–609.

[26] M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study on the bug-proneness of different types of code clones," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 91–100.

[27] V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ó Cinnéide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 932–961, 2020.

[28] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and nlp to promote a consistent usage of identifiers," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017, pp. 81–90.

[29] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 369–378.

[30] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter, "Design recovery and maintenance of build systems," in *2007 IEEE International Conference on Software Maintenance*, 2007, pp. 114–123.

[31] J. O'neal, K. Weide, and A. Dubey, "Experience report: refactoring the mesh interface in flash, a multiphysics software," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, 2018, pp. 1–6.

[32] M. Aziz Parande and G. Koru, "A longitudinal analysis of the dependency concentration in smaller modules for open-source software products," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–5.

[33] T. D. Oyetoyan, D. S. Cruzes, and C. Thurmann-Nielsen, "A decision support system to refactor class cycles," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 231–240.

[34] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 357–366.

[35] B. Cyganek, "Adding parallelism to the hybrid image processing library in multi-threading and multi-core systems," in *2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications*, 2011, pp. 1–8.

[36] R. Hardt and E. V. Munson, "An empirical evaluation of ant build maintenance using formiga," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 201–210.

[37] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y.-G. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–5.

[38] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria, "An industrial experience report on performance-aware refactoring on a database-centric web application," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 653–664.

[39] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[40] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao, "Incremental and iterative reengineering towards software product line: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 418–427.

[41] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 104–113.

[42] A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts, "Reducing code complexity through code refactoring and model-based rejuvenation," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 617–621.

[43] I. Griffith, S. Wahl, and C. Izurieta, "Evolution of legacy system comprehensibility through automated refactoring," in *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, 2011, pp. 35–42.

[44] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Behind the intent of extract method refactoring: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 668–694, 2024.

[45] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of legacy software using model learning and equivalence checking: An industrial experience report," in *Integrated Formal Methods*, E. Ábrahám and M. Huisman, Eds. Cham: Springer International Publishing, 2016, pp. 311–325.

[46] J. Park, M. Kim, and D. H. Bae, "An empirical study of supplementary patches in open source projects," *Empirical Software Engineering*, vol. 22, pp. 436–473, 2017.

[47] T. L. Nguyen, A. Fish, and M. Song, "An empirical study on similar changes in evolving software," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018, pp. 0560–0563.

[48] B. L. Sousa, M. A. Bigonha, and K. A. Ferreira, "An exploratory study on cooccurrence of design patterns and bad smells using software metrics," *Software: Practice and Experience*, vol. 49, no. 7, pp. 1079–1113, 2019.

[49] F. Schmidt, S. G. MacDonell, and A. M. Connor, *An Automatic Architecture Reconstruction and Refactoring Framework*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 95–111.

[50] O. Mehani, G. Jourjon, T. Rakotoarivelo, and M. Ott, "An instrumentation framework for the critical task of measurement collection in the future internet," *Computer Networks*, vol. 63, pp. 68–83, 2014.

[51] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen, "An evaluation of clone detection techniques for crosscutting concerns," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 200–209.

[52] Y. Kosker, B. Turhan, and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Systems with Applications*, vol. 36, no. 6, pp. 10 000–10 003, 2009.

[53] V. Alizadeh, H. Fehri, and M. Kessentini, "Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 181–192.

[54] H. Altrichter, S. Kemmis, R. McTaggart, and O. Zuber-Skerritt, "The concept of action research," *The Learning Organization*, vol. 9, no. 3, pp. 125–131, 2002.

[55] O. Zuber-Skerritt, *Action Learning and Action Research: Paradigm, Praxis and Programs*. Publisher Unknown, 2001, vol. 1, no. 20, pp. 1–27. [Online]. Available: https://api.semanticscholar.org/CorpusID:140798095

[56] C. D. Yangyang Sun and G. Feng, "Optimal versioning strategies for software firms in the competitive environment," *International Journal of Production Research*, vol. 59, no. 22, pp. 6881–6897, 2021. [Online]. Available: https://doi.org/10.1080/00207543.2020.1828641

[57] C. Katsanos, N. Tselios, N. Avouris, S. Demetriadis, I. Stamelos, and L. Angelis, "Cross-study reliability of the open card sorting method," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3290607.3312999

[58] J. Van Geet, P. Ebraert, and S. Demeyer, "Redocumentation of a legacy banking system: an experience report," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 33–41. [Online]. Available: https://doi.org/10.1145/1862372.1862382

[59] L. von Kurnatowski, D. Heidrich, N. Güden, A. Schreiber, H. Polzin, and C. Stangl, *Analysing and Visualizing large Aerospace Software Systems*. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2021-4082

[60] A. Solomon and Z. Crawford, "Transitioning from legacy air traffic management to airspace management through secure, cloud-native automation solutions," in *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, 2021, pp. 1–8.

[61] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza, "Modernizing legacy systems with microservices: A roadmap," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 149–159. [Online]. Available: https://doi.org/10.1145/3463274.3463334

[62] T. Tervoort, M. T. De Oliveira, W. Pieters, P. Van Gelder, S. D. Olabarriaga, and H. Marquering, "Solutions for mitigating cybersecurity risks caused by legacy software in medical devices: A scoping review," *IEEE Access*, vol. 8, pp. 84 352–84 361, 2020.

[63] Alqoud, Abdulrahman, Schaefer, Dirk, and Milisavljevic-Syed, Jelena, "Industry 4.0: a systematic review of legacy manufacturing system digital retrofitting," *Manufacturing Rev.*, vol. 9, p. 32, 2022. [Online]. Available: https://doi.org/10.1051/mfreview/2022031

[64] T. Verhoeff, "Staying dry with oo and fp," *Olympiads in Informatics*, vol. 18, pp. 113–128, jul 2024.

[65] A. A. Eduardo, R. A. d. Sousa, R. M. Loureiro, A. Tachibana, and A. P. dos Santos, "Exploring the kiss principle (keep it simple) for decision support systems," *medRxiv*, 2022. [Online]. Available: https://www.medrxiv.org/content/early/2022/08/25/2022.08.23.22279126