# The Driving Forces of API Evolution

William Granli and John Burchell
Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
william.granli@gmail.com,
john.a.burchell@gmail.com

Imed Hammouda and Eric Knauss
Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
{imed.hammouda,eric.knauss}@cse.gu.se

## ABSTRACT

Evolving an Application Programming Interface (API) is a delicate activity, as modifications to them can significantly impact their users. The increasing use of APIs means that software development organisations must take an empirical and scientific approach to the way they manage the evolution of their APIs. If no attempt at analysing or quantifying the evolution of an API is made, there will be a diminished understanding of the evolution, and possible improvements to the maintenance strategy will be difficult to identify. We believe that long-standing software evolution theories can provide additional insight to the field of APIs, and can be of great use to companies maintaining APIs. In this case study, we conduct a qualitative investigation to understand what drives the evolution of an industry company's existing API, by examining two versions of the API interface. The changes were analysed based on two software evolution theories, and the extent to which we could reverse engineer the change decisions was determined by interviewing an architect of the API. The results of this analysis show that the largest driving force of the APIs evolution was the desire for new functionality. Our findings which show that changes happen sporadically, rather than continuously, appear to show that the law of Conservation of Organisational Stability was not a considerable factor for the evolution of the API. We also found that it is possible to reverse engineer change decisions and in doing so, identified that the feedback loop of an API is an important area of improvement.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*Restructuring, reverse engineering, and re-engineering*

## Keywords

API Design, Software Evolution, Software Maintenance



*IWPSE'15*, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804364

## 1. INTRODUCTION

As the software industry and the open-source movement continue to grow, the number of public Application Programming Interfaces (APIs) is steadily increasing. APIs can improve the development speed [28], contribute to higher quality software [28] and increase the reusability of software [2]. There is a consensus in that modifications to APIs could negatively impact the users of the API [5, 14, 20, 23]. The main reason being that API users are required to update their application code, thus causing a disruption in the application's software ecosystem [21]. In the discipline of software evolution, such updates to software are studied from an evolutionary standpoint. Software can be updated for different reasons and these motives can be grouped into corrective, adaptive, perfective and preventive changes [19].

Modifications to deployed APIs may negatively impact its users. It is therefore important to investigate why these changes occur. Understanding the motives behind the changes could help API architects to prevent potential future changes to their APIs. Before industry can safely include such information in their feedback loop, we must critically assess to what degree we can reverse engineer such motives. Identifying the types of changes that occur when an API evolves will improve communication and discussions regarding maintenance and evolution tasks for the case company. This improved communication could provide a common language that developers, users and management can use to discuss potential evolution of the APIs. Understanding the evolution theories could further assist understanding of how change will occur and how to plan for it. Lastly, by keeping a history of changes to an API over time, trends can be identified which could help with future API design.

Previous studies have investigated programming language APIs, libraries or frameworks focusing on causes and effects of modifications to APIs [10, 15, 27]. However, to the best of our knowledge, no studies have been performed on embedded platform APIs from a software evolution perspective. Our study will fill this gap by analysing an embedded platform API, classifying the types of changes that occur and analysing them with the use of software evolution theories.

The goal of the study was twofold. Firstly, the aim was to explore what are the driving forces of API evolution and secondly, to validate the results' level of correctness by comparing our findings with the case company's explanation of the changes. With this information, we could determine to what extent these change decisions can be reverse engineered.

The sections following the Introduction are structured as follows: In Section 2, we present the background and previously published work that is related to our study and introduce the case company and their API. A description of our methodology is introduced in Section 3. In Section 4, the results of our study are presented and in Section 5, we discuss the implications of our work. In Section 6 we summarise the study and give suggestions for future work that builds on our research.

## 2. BACKGROUND

This section will introduce the fields of APIs, software evolution and it will provide a review of studies that are related to our study. The subsequent section will introduce the case company with a description of the API used in this study.

### 2.1 APIs

An API offers an interface through which developers can access programmatic functionality. APIs can be seen as having three interacting layers; the public interface, the actual implementation of the functionality and an intersecting layer which provides utilities such as error handling, third party libraries and additional auxiliary features. Typically, the interfaces provide the definitions of functions and data structures while the implementation realises those interfaces.

API design is notoriously difficult, as a myriad of design and performance decisions must be taken into consideration when creating APIs [2, 6, 28]. Examples of such design decisions include how to structure an object's constructor parameters or if the API should display errors at compilation or at runtime [28]. More trivial design problems, such as assigning names to API features or correctly naming user-defined types, can have a significant impact on the usability of an API [27]. When facing such design challenges, the following four factors are important to consider: a) The API must be understandable through good documentation, b) the API must not be overly abstract, c) the API must be reusable and d) the API must be easy to learn [27]. One of the most important qualities in an API is that the intent of the API must be clear to the user [27, 28]. The design decisions reached during development of an API will affect the overall usability of the API. Measuring such an effect can be done by investigating the twelve cognitive dimensions that are impacted by interactions between the API and its users [8].

### 2.2 Software Evolution

Software evolution is a field that studies the application of software maintenance activities, changes in software processes and the resulting, evolved versions of the software. The concept of software maintenance has existed since the 1960s when it was first introduced to the software development community [19].

A set of four categories describing different kinds of software maintenance [19] became the basis upon which twelve new types were developed [7]. The twelve types of software evolution and software maintenance, as seen in Table 1, describe a software evolution activity that relates to one of three particular areas; the code, the software and the customer-experienced functionality.

| Type | Explanation |
|---|---|
| Enhancive | Inclusion of new functionality |
| Corrective | Corrections of functionality |
| Reductive | Reductions of functionality |
| Adaptive | Inclusion of new technology |
| Performance | Improvements to performance |
| Preventive | Improvements to future maintainability |
| Groomative | Improvements to maintainability |
| Updative | Updates to documentation |
| Reformative | Changes to documentation |
| Evaluative | Inspection activities |
| Consultive | Consultations activities |
| Training | Training activities |

**Table 1: Types of Software Evolution [7]**

1. **Continuing Change**
   E-Type systems must be continually adapted else they become progressively less satisfactory.
2. **Increasing Complexity**
   As an E-Type system evolves its complexity increases unless work is done to maintain or reduce it.
3. **Self Regulation**
   E-Type system evolution process is self regulating with distribution of product and process measures close to normal.
4. **Conservation of Organisational Stability**
   The average effective global activity rate in an evolving E-Type system is invariant over product lifetime.
5. **Conservation of Familiarity**
   As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
6. **Continuing Growth**
   The functional content of E-Type systems must be continually increased to maintain user satisfaction over their lifetime.
7. **Declining Quality**
   The quality of E-Type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
8. **Feedback System**
   E-Type evolution process constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

**Table 2: Lehman's Laws of Software Evolution [18]**

Software evolution theory suggests that software programs can be grouped into three different types of systems, namely S-programs, P-programs and E-programs [17]. S-programs are programs which are exactly derivable from a specification. An example of such a program is one which finds the lowest common denominator for two integers. A key characteristic of an S-program is that if its functionality is changed, the end product will be a new program and not an evolved version of the previous program, since the mapping between specification and program is exact. P-programs are programs which are inherently more complex, and "solve a problem which can be precisely formulated, but whose solution must inevitably reflect an approximation of the real world". Furthermore, P-programs are programs which are centred around the perception of users, analysts or programmers. This perception is often commonly a mathematical model of the problem. An example of such a program is a program which predicts weather, since it is derived from a set of hydrodynamic equations. E-programs are described as "inherently even more change prone [than S-programs and P-programs]. They are programs that mechanize a human or societal activity". These factors can be explained by the difference between P-programs and a E-programs which is that E-programs "change the very nature of the problem to be solved" by "becoming a part of the world it models". Further, a characteristic of E-programs is that, during development, the specification must be developed by involving predictions of the consequences of the deployment of that system will have, since once the program is deployed "questions of correctness, appropriateness, and satisfaction arise which inevitably leads to additional pressure for change". Lehman's eight laws of software evolution are said to apply to such E-programs. The laws are presented in Table 2.

## 2.3 API Evolution

The analysis of APIs in the context of software evolution has primarily focused on APIs that are part of large programming languages, such as Java [15, 27] and Smalltalk [23]. Attempts to uncover the intents of the changes made in the Java libraries, AWT and Swing, have also been undertaken [15], leading to suggestions that the use of a strong architecture is vital for ensuring the successful evolution of an API [15]. Investigations of three frameworks and one library indicated that 80% of refactoring changes to APIs negatively affected existing applications [10]. The changes, sometimes referred to as ripple effects [23] indicate that negative effects of changes can propagate throughout a software ecosystem. It was found that 14% of non-trivial API deprecations caused errors in at least one project, with the worst case of 79 projects being affected [23].

## 2.4 Case Company Description

The case company of this study will from here on be referred to as *company A*. Company A operates in the domain of video surveillance and has offices in 49 countries worldwide. Company A is the global market leader in the markets of network cameras and video encoders. They develop embedded software for security cameras. The cameras are designed to be accessible through APIs that are developed and maintained by the company.

The API analysed in this study is written in the programming language C and has been deployed for several years. It was recently updated from v1.4 to v2.0, which is the current active version. The changes between these two versions are what has been analysed in this study. Between the release of v1.0 and v1.4, no functionality was added, as it only added additional build options for other hardware platforms. The API is used by company partners, which are other companies developing applications for the cameras. These applications are, in turn, used by the end-users of the cameras. The API is compatible with a wide range of camera types that offer different functionality and are used for different purposes.

### 2.4.1 APIs as Systems

Company A's API is rather complex, and in addition to the API interface and implementation code, it includes a middle-layer which manages error handling and the access to various 3$^{rd}$ party libraries. In this study, we use the API interface and the middle-layers as units of analysis. Examples of where the responsibilities exceed those of a primitive API interface is that it includes functionality such as error handling and a daemon which facilitates communication between hardware entities. In addition to that is an SDK which includes the API documentation, user manuals and example applications that utilise the API code. Based on these characteristics and the IEEE definition of a system, which reads as follows; "A combination of interacting elements organised to achieve one or more stated purposes...Organised to accomplish a specific function or set of functions within a specific environment" [1], we consider the API to be a system because of the combination of the interacting parts of the API, coupled with the specific nature of the problem the API is addressing.

Building on this, we regard the API examined in this study to be an E-type system based on that it is a part of the world it models. This is evident from that it is a vital part of the cameras' and the only option for developing software for the cameras. The API is also considered by company A to be a vital part of the cameras' software ecosystem. This factor corresponds with the characteristic of E-type systems which relates to the system being susceptible to change from pressure that occurs after the system has been deployed. Similarly, we do not consider the API to be a P-type system, based on the fact that the system is virtually impossible to model due to its high dependence on the environment in which it operates in.

# 3. METHODOLOGY

This study has been conducted using the case study methodology [25]. Our research can be classified as an embedded case study [30], since both the API interface and API documentation have been used as units of analysis. The reason the selected methodology was used is that it is essential to study the phenomenon of API change in its natural context. The applicability of case studies in such scenarios is supported by existing literature [4, 24, 25, 30]. An alternative approach that was considered was design research, but if a prototype API was to be used instead of one which is tried and tested in an industry setting, the study would lack real-life context [25]. An additional motivation for why the case study approach was used is that there is little existing research conducted in the area of drivers of API change and that input from the industry is vital to the success of the research.

**Phase 1: Data Collection**

1. Inspect v1.4
2. Inspect v2.0
3. Identify changes between versions
4. Conduct interview
5. Structure gathered data
6. Formulate hypotheses

**Phase 2: Data Analysis**

1. Code changes
2. Group changes
3. Categorise groups
4. Identify and formulate trends
5. Analyse based on types [7]
6. Analyse based on laws [18]
7. Validate hypotheses

**Table 3: Methodology Overview**

The study was conducted in two phases. The aim of the first phase was to generate hypotheses and to formulate clear research questions. This was achieved by using a data-driven approach to analysing the interface code and documentation. The second phase included analysis of the interviews, during which we aimed to confirm the hypotheses that were formulated in the first phase. Both phases were performed iteratively. This created the opportunity to improve the data analysis as the study progressed, as well as, allowing us to adapt to possible changes in direction, due to the hypothesis generating approach of the first phase. An overview of the methodology can be found in Table 3.

The process used for examining the code and documentation was inspired by grounded theory analysis [26], as it is recommended for hypothesis generating studies [25, 26]. The exploratory analysis lead us to formulate the following research questions.

**RQ1** What drives API evolution?

**RQ2** To what extent can we reverse engineer API change decisions?

## 3.1 Data Collection

The data were collected primarily through inspections of the source code and of accompanying documentation. The source code was comprised of the two API versions previously mentioned. Each version of the API had its own respective documentation, including example code, a library description and a basic development manual.

The API interface code was inspected sequentially, starting with v1.4. Both versions of the API underwent the same type of inspection. This involved extracting method signatures, enums, structs, typedefs and macros, which were subsequently stored in spreadsheets. The extraction involved a lot of manual work, but Git's [12] diff command was used to identify changes in files which exist in both versions. After a change was identified in the code, a description of the corresponding change in the documentation was added to the spreadsheet. The module and file in which the change was identified was then logged together with an ID and a description of the change. The data collection from code and documentation was performed independently by both researchers to reduce the risk of human errors affecting the results. If the data collected by each researcher differed, the cause of the discrepancy was investigated and resolved.

After the initial source code and documentation inspection, an interview was conducted. The main purpose of the interview was to validate our findings from the code inspection with the API architect. The interview also served the purpose of providing additional insight into what drove the evolution of the API, by complementing what was found from the inspections. The final reason for interviewing the API architect was to gather information about the API users and how their decisions, needs and requirements have affected the development of the API.

The interview was conducted using a semi-structured approach [24]. Structure to the interview was provided by organising it around the software evolution theories [7, 18]. Investigative and open-ended questions related to each type and law were asked to provide a basis for comparison between the answers and our initial analysis. The interviewee was encouraged to speak freely, even if it required a change in direction or topic. This was encouraged in order to fulfil our exploratory goal of the study.

## 3.2 Data Analysis

To analyse the data, each change identified in the spreadsheet was coded based on which module the change occurred in and based on if the change added, removed, modified or did not affect functionality. Grouping the codes by module allowed us to form general concepts of change. However, changes that affected multiple modules were classified on their own. These concepts were then grouped into categories which were based on common patterns between the concepts. An example of a pattern found during the analysis was that several concepts followed the pattern of adding new functionality. These categories were then used to identify the most notable trends in the evolution of the API. These trends were later used as a basis for discussion during the interview.

The trends identified previously were then classified with the help of the decision tree used for classifying the types of software evolution and software maintenance [7]. Each of the previously identified trends were given a main type of change and, if appropriate, a secondary type. Main types of change identify the primary motive for a trend, whereas the secondary types are additional motivations for a trend. For example, if an identified trend was the re-implementation of an old module that included new functionality and made use of a new interface, it would be classified as mainly enhancive and secondarily groomative. In addition to this, we analysed the trends based on Lehman's Laws, to determine if they applied to the changes made to the API. This information, as well as the results of the interview, were then used as the

basis for answering **RQ1**.

To allow us to answer **RQ2**, we compared our analysis and interpretation of the trends identified prior, against what was expressed by the API architect, during the interview. Success was determined by comparing the inter-rater reliability, by calculating Cohen's kappa [9] for the gathered data.

## 3.3 Validity Threats

In this section we discuss possible threats to construct validity, internal validity, external validity and reliability [25].

**Construct validity** Since our study is largely based on existing theories [7, 18], the validity of it directly correlates to the validity of the theories and the applicability of them. With regards to Lehman's laws of software evolution, the laws are said to only apply to E-type systems. According to our analysis, we consider the API to be of the E-type, but if our interpretations of Lehman's definitions are incorrect, it could be considered a threat to the construct validity of our study.

**Internal validity** Limitations related to internal validity have been acknowledged by analysing the code and documentation jointly. This has contributed to triangulating the results and discovering possible inconsistencies. Further triangulation of the results was performed by interviewing an architect of the API. Business-related factors might have had an impact on the evolution of the API, and these were not closely investigated. This was not in the scope of the study and would best be investigated in a study complementary to ours. The study revealed that the API users were significant to the evolution of the API. Since API users were not interviewed, it might be an affecting factor which was left unexplored.

**External validity** Since only two versions of the API were analysed, we cannot claim any generalisability for our study. We do, however, believe that our study can offer comparability for companies that chose to do a similar analysis on their own APIs.

**Reliability** Due to restricted access to company A's source code, we were not able to include versions prior to v1.4 in our analysis. This means that the evolution of the API was studied during a rather short period of its lifetime. It was also not possible to analyse the implementation of the API, something which might have contributed to more accurate predictions of the motives behind the changes. We consider these factors to be the remaining threats to reliability. Reliability has been increased by conducting the study according to an accredited guide to case studies [25]. In addition to that, strategies from additional well-established papers [3, 24, 26] have been used to increase the trustworthiness of the data gathering and data analysis.

# 4. RESULTS

This section presents the results of our study and is structured around the research questions. Each section begins with a summary of the results and a subsequent in depth description of the findings.

In total, thirteen unique changes were identified when analysing the source code, the majority of which were additions of functionality of the API. Such additions include added functionality to allow mechanical and digital control of the camera, to utilise additional storage devices, to allow serial connectivity and to allow audio analysis on the cameras. Changes that were pure additions to the API were not attributed with any secondary types of change.

| Type | Main Types | Secondary Types |
|---|---|---|
| Enhancive | 7 | - |
| Corrective | - | - |
| Reductive | 1 | - |
| Adaptive | 1 | - |
| Performance | - | - |
| Preventive | 2 | - |
| Groomative | - | 5 |
| Updative | 1 | 1 |
| Reformative | 1 | 1 |
| Evaluative | - | - |
| Consultive | - | - |
| Training | - | - |

**Table 4: Identified Types of Change**

Many of the changes were deprecations that were later re-implemented as new modules. Changes of this kind were grouped into a single type instead of two. Such examples include the deprecation and re-implementation of an event system, dynamic web page generation and configuration utilities. Changes of this kind often had an accompanying secondary type of change.

The remainder of the changes were defined as general trends. These changes were not changes to individual parts of the API, but were instead broader changes. Such examples include a shift towards using interfaces to access features, a different error-handling approach and the inclusion of additional 3$^{rd}$ party libraries. Typically, these general trends also have a secondary type of change attributed to them.

## 4.1 The Driving Forces of API Evolution

The results show that certain types of change are more prevalent than others, these results are shown in Table 4. Analysis of the results showed that enhancive changes were the primary types of change that drove the evolution of this API and that they acted as a catalyst for many of the other changes in the API. Updative changes were found to be consequences of enhancive changes, rather than being drivers of evolution themselves. Supporting the users of the API through the combination of preventive and groomative types of change, together formed another strong driver of the API's evolution. Reductive and adaptive changes, while important, did not strongly drive the evolution of the API. No evidence was found for corrective or performance types of change. Similarly, there was no evidence found for evaluative, consultive or training activities.

Our analysis of the results related to **RQ1** showed that seven out of the eight of Lehman's laws [18] are present in the API. A summary of our findings can be seen in Table 5. The evidence found for the laws of Continuing Change, Self Regulation, Conservation of Familiarity, Continuing Growth and Feedback System mainly relate to enhancive changes that add functionality to the API. It was also found that the laws of Increasing Complexity and Declining Quality do

| Law | Applicability |
|---|---|
| Continuing Change | True |
| Increasing Complexity | True |
| Self Regulation | True |
| Conservation of Organisational Stability | False |
| Conservation of Familiarity | True |
| Continuing Growth | True |
| Declining Quality | True |
| Feedback System | True |

**Table 5: The Applicability of Lehman's laws**

apply, but that the changes which support these laws usually had a secondary intent related to complexity or quality. The only law which was found to have little effect in the evolution of the API analysed in this study was the law of Conservation of Organisational Stability.

Following this section, we will discuss the results related to each type of change category, discussing the results for each of them with examples from the changes identified between the two versions of the API. Subsequently, each of Lehman's laws will be discussed and our results will be presented showing if the laws were observed.

### *4.1.1 Types of Change*

**Enhancive** Additions or re-implementations of functionality are defined to be enhancive changes. In addition to this, we have included deprecations that have then been re-implemented to also be enhancive. Seven of the identified changes were given the type of enhancive. The majority of which were additions of new functionality to the API. Two examples of added functionality are the additions of a camera movement module and an audio analysis module. One example of a deprecation that was re-implemented is the event system. Given that over half of the changes were identified to be enhancive, we conclude that the APIs evolution primary type of change was enhancive.

**Corrective** Corrective changes are identified as types of change that fix broken functionality. No corrective changes were identified between v1.4 and v2.0. There are two main reasons that we did not find any changes of this type. Firstly, the majority of changes were additions or re-implementations of functionality. This meant that most of the existing code was either removed, replaced or deprecated, resulting in little to no obvious fixes. Secondly, we only inspected public facing API and not the implementation itself. We therefore conclude that the evolution of the API was not driven by corrective changes.

**Reductive** Reductive changes are defined as changes that remove or reduce functionality. This is distinctive from a deprecation as reductive types of change remove functionality completely. Only a single reductive type of change was identified. This reductive change removed functionality that provided the ability to buffer individual images for closer analysis by a user. Given that it is more common to refactor APIs and not remove functionality [10, 29], this result was not surprising. It was revealed during the interview that the functionality was removed because company A no longer wished to support this feature. As this was the only reductive change found, we conclude that the evolution of the API was not primarily driven by reductive changes.

**Adaptive** Adaptive changes are defined to be changes which involve changes to technology or resources used. A single change was identified as being adaptive; the inclusion of new 3$^{rd}$ party libraries. Libraries were added for audio and data communication with the pre-existing low-level library being incorporated into the API. We therefore conclude that the evolution of the API has not been primarily driven by adaptive changes.

**Performance** Performance changes are defined to be types of change that intentionally alter system performance. No evidence of performance-driven changes were identified in the API. This therefore leads us to conclude that the API's evolution is not driven by performance changes.

**Preventive** Preventive changes are defined to be changes that attempt to avoid future maintenance. This should not be confused with changes that attempt to make the current state of the system more maintainable. Two changes were identified to be preventive types of change. A new error handling system was one of these changes. v2.0 of the API added specific error handling modules and functions which replaced the pre-existing error-handling in v1.4 of the API. The redesign of the system to make use of interfaces was also classified as a preventive change. Using interfaces to access modules helps to make them more maintainable and more easily extensible. Preventive types of change appear to drive the evolution this API more strongly than other types of change. This could be due to the fact that as the API grows, the need for higher maintenance also increase, a fact that also is supported by Lehman's laws.

**Groomative** Groomative changes are defined to be changes that aim to immediately make the code more maintainable. While none of the changes identified were primarily groomative changes, three of them were secondarily classified as being groomative. This indicates that the groomative nature of a change is often a positive affect of other types of change. The move towards using interfaces can be seen as an example of this. While the primary motive of change was to make the API more maintainable in the future, making the API more maintainable immediately was also a reason for the change. While not a primary factor for any of the changes, groomative types of change should still be viewed as important to the API's evolution.

**Updative** Updative changes are defined to be changes to documentation of a system that are made to conform with changes in the code. A single change was determined to be updative; updating the API documentation to include the new functionality of v2.0. The updates included a new API specification, deprecation list and the inclusion of the new features and libraries. Updative changes have very little impact on the functionality of the API yet they are still important changes for the usability of the API [27]. Changes to the API cause the need for updative changes to its documentation, without them, the usability of the API could suffer. We therefore conclude that the evolution of this API was not driven by updative changes.

**Reformative** Reformative changes are defined to be changes that update the documentation to the stakeholders needs. An example of such a change is that the structure of the documentation is changed. One reformative change was identified between v1.4 and v2.0; a redesign of the existing example programs. The aim of this change was to update the examples to illustrate the new functionality in the API, showing how the modules interact and at the same time, presenting a standard for how to use the API. We therefore conclude that reformative changes, while important for the usability of the API, did not drive the evolution of the API.

**Evaluative, Consultive and Training** Evaluative, consultive and training are all types of activities, rather than changes that a system can undergo. Evaluative activities are defined as activities such as auditing or evaluating the software. Similarly, consultive activities involve consultations with customers about the software. Finally, training activities involve training for customers and users of the software. None of the aforementioned activities occurred as part of the APIs evolution. We can therefore conclude that these change types were unimportant for the evolution of the API.

### *4.1.2 Lehman's Laws*

**Continuing Change** Company A described that including new functional content was one of the top priorities for the release of v2.0. This is supported by the characteristic of the changes identified in this study. Four of the thirteen changes were additions of new modules, and additional functionality was also added to re-implemented modules. It was also mentioned that the API users had previously requested more frequent updates to the API, despite the drawbacks this could cause. One of the main reasons for continuously adapting the API is also to control the way API users use the API. Prior to v2.0 being released, a number of API users used workarounds to implement applications which contained functionality that was not yet offered by the API. These applications were fully functional, but there existed no standard method of implementation. We therefore conclude that the law of Continuing Change was a driving factor for this API.

**Increasing Complexity** A majority of the changes introduced in v2.0 aimed to improve the structure and the way that the API was used. Many of these changes were related to the same design choices, which enforced a certain implementation style on the whole API. One example of this is the change to the interfaces which set a standard for the whole API. During the interview it was also concluded that consistency and following certain standards is an important factor to consider when designing APIs. It was also made clear that a contributing factor for changing certain parts of the API was to bring it into line with the new interface design. The interviewee mentioned that one way of achieving this was to follow guidelines or design best practices, but that company A currently did not make use of these. Although the importance of reducing the complexity was stated, it was made clear during the interview that one module was not updated between the versions, even though it did not follow the new standards of v2.0. In conclusion, we establish that the law of Increasing Complexity was a driving factor for this API.

**Self Regulation** The release of v1.4 did not include any changes related to the API interface. v1.0 through to v1.4 strictly added support for additional build target platforms. When comparing the changes introduced in v2.0 to the ones in previous versions, it is clear that the growth of the API has gradually increased. During the interview it was also mentioned that the future plans for the API did not include any major changes and that only a few minor changes were planned in the near future. The future updates to the API will also be made incrementally and there will not be a big bang update similar to that of v2.0. This supports that the law of Self Regulation was significant in the API's evolution, assuming that v2.0 can be considered to be the peak of the normal distribution curve of the APIs growth.

**Conservation of Organisational Stability** The changes to the API have been made very sporadically, where only two new versions have been released since the deployment of v1.0. As mentioned in Section 4.1.2, the changes introduced in v1.4 did not involve any changes to the existing API functions. Consequently, the only significant update to the API was made in the transition to v2.0. The internal development of the API has also been made sporadically and there have been periods where the API has not been actively developed. This suggests that the law of Conservation of Organisational Stability was not a considerable factor in the evolution of the API.

**Conservation of Familiarity** The documentation that is included with the API was rigorously updated with the introduction of v2.0. All functionality which was added or modified in v2.0 was updated accordingly in the documentation. The same pattern can be seen for the example code, which went through major changes as a result of the changes to the API interface. Company A's intent to control the way that their API is used also suggests that the knowledge of how to use the API is of importance, since if the API is used in too many different ways, it will be more difficult for company A to have relevant documentation and example code. It was also expressed during the interview that further transparency in how the API should be used could be achieved. One example mentioned of how this could be achieved is by improving the expressiveness regarding which API functions are supported on which hardware types. These facts support that the law of Conservation of Familiarity played an important part in the evolution of the API.

**Continuing Growth** Large parts of the changes between the versions were strictly additions of new functional content. 4 new modules were added and significant additions of new functionality were added to the modules which were re-designed from v1.4. The interviewee also made it clear that increasing the functionality was one of the main goals with the 2.0 update. This also ties in with the ambition of controlling the way that the API is used, as mentioned previously. It was also mentioned that company A had received feedback from the API users that more frequent updates to the functional content was desirable, even though this may require them to update their application code. We therefore conclude that the law of continuing growth was a driving force for the evolution of the API.

**Declining Quality** The interviews established that a groomative motive was not the primary reason for implementing any change. The changes to the example code and the documentation, as well as the re-designed modules are examples of these, where the main motive behind the change was to either prevent unwanted usage or to increase the functional content. These changes did, however, also increase the qual-

ity and were a substantial part of the update. Our analysis also shows that changes related to quality are bound to happen, as other types of changes often incorporate quality related aspects. We therefore settle that the law of Declining Quality was a driving force for the evolution of the API.

**Feedback System** One of the future goals of improvement of company A is to better include the API users and end-users of the applications in their feedback loop. Currently, only the API users are included in this feedback loop and the API architect expressed the desire to also include the end-users. The reasoning for this was to increase the quality, and to be able to deliver content which is useful for the users and which leads to good applications being developed. Prior to v2.0 being released, an example of how the API users were included in the feedback loop was when they implemented functionality related to sound, despite the fact there was no interface for this. The experiences of this API user, were later considered when developing the axsound module. One benefit of having a more structured feedback loop, would be that the API developers would have greater awareness of potential errors in the implementation of the API. Another ambition which was mentioned, was to bring scientific theories and best practices from research into consideration when developing the API. This would aim to complement the current strategy which is largely based around the expertise and experiences of the API architects. Based on these factors and clearly expressed ambitions, we conclude that the law of Feedback System was of significant importance during the evolution of the API.

## 4.2 Reverse Engineering Change Decisions

After cross-examining our initial analysis with the interviewee's responses, we found that 85% of our main type predictions matched the actual motive of the change and that 85% of our secondary type predictions were correct. The kappa value [9] for the predictions was 0.812, which shows that there is a high level of agreement between the predictions and the change decisions. In Table 6, a detailed view of each prediction juxtaposed against the actual change decision can be found for both the main and secondary types.

The changes related to functionality being added, modified (deprecated) or removed were predicted with a 100% accuracy. These changes were deemed the most significant by the interviewee, and it was expressed during the interview that the these types of changes were the main driving factor of the development of the new version. Contrary to this, three out of five changes which were unrelated to functionality were accurately predicted. The two changes which were not predicted accurately were inaccurate because the main type was confused with the secondary type. This shows that, although the prediction was incorrect, the general reasoning was correct. The changes which had no secondary type were all predicted accurately, which further shows that changes with several types can be diffuse.

| Main | | Secondary | |
|---|---|---|---|
| **Prediction** | **Actual** | **Prediction** | **Actual** |
| Enhancive | Enhancive | N/A | N/A |
| Enhancive | Enhancive | N/A | N/A |
| Enhancive | Enhancive | N/A | N/A |
| Enhancive | Enhancive | N/A | N/A |
| Enhancive | Enhancive | Groomative | Groomative |
| Enhancive | Enhancive | Groomative | Groomative |
| Enhancive | Enhancive | Groomative | Groomative |
| Reductive | Reductive | N/A | N/A |
| Adaptive | Adaptive | N/A | N/A |
| Preventive | Preventive | Groomative | Groomative |
| Updative | Updative | Reformative | Reformative |
| Updative | Reformative | Reformative | Updative |
| Groomative | Preventive | Preventive | Groomative |

**Table 6: Categorisation of Changes**

# 5. DISCUSSION

RQ1: **What drives API evolution?**

Our results show that the major driving force for change in the case company's API was the call for increased functionality. The classification of the types of software evolution and software maintenance showed that enhancive changes which either aimed to add modules or new functionality to existing modules were the most common. We also think that it is interesting to note that a majority of the changes which did not affect functionality, such as the updative changes, were spurred because of previous changes that were related to functionality. Following this line of thought, the desire for new functionality can be seen as a catalyst that allows new change to occur. The view of considering functionality to be a major driver for change is supported by the analysis with Lehman's laws. In the law of Continuing Change, Lehman concludes that "an E-type system must be continually adapted or it becomes progressively less satisfactory". The results of our work support this view, as the changes did not only add functionality, but it was also found that the application developers most major concern were updates to functionality.

In relation to this, previous work has identified that refactorings make up a substantial part of the changes APIs undergo [10, 11, 13, 29]. These results are not in line with ours, since our results show that groomative changes are uncommon and that changes related to quality are not primary drivers of change, rather, they are ripple-effects of changes to functionality [23].

The results also showed that usability was the second-most important driver of the studied API's evolution. The classified updative, groomative and reformative changes all affected the maintainability and usability of the API and involved activities that have been found to improve the usability and maintainability of an API [2, 8, 22, 27]. The interviewee discussed how company A wanted to aid developers creating applications by improving the interfaces that the developers are using. This improvement to the interfaces would better communicate the design and intentions to the developer. Improving the documentation of the API was also important for v2.0, as were improvements to the example code. These updates were intended to aid developers and create a set of standards that should be followed when creating applications. These changes correspond with three of Lehman's laws: Conservation of Familiarity, Declining Quality and Feedback Loop.

Our results found that seven out of eight of Lehman's laws affected the evolution of the studied API. The law of Conservation of Organisational Stability was found to have less significance, since changes to APIs are unfavourable due to the potential disruption the applications using the API could face. This inertia to change in API evolution is supported by other research, which also conclude that the negative effect that changes to APIs may have is a major factor to be considered [5, 14, 20, 23].

Based on our results, we form the conclusion that the desire for functionality and the reluctance of changing are two opposing forces that caused updates to this API to be infrequent. In the evolution of APIs, the negative effects of changes are greater than in other types of systems, which is the main indication that the law of Conservation of Organisational Stability is less significant for this API. This occurrence has been considered by Lehman, as he noted that "the laws are not immutable, since they arise from the habits and practices of humans" [17]. The underlying reason for this is that the laws are influenced by conscious thought processes, based on human understanding [16], which, in the case of this API, is the conscious decision to limit updates as much as possible.

RQ2: **To what extent can we reverse engineer API change decisions?**

We conclude that we were able to reverse engineer the change decisions to a large extent. This is clear from the results, which showed that 85% of the changes were reverse engineered accurately. The kappa value of 0.812 further supports this. We interpret that the significance of this result suggests that reverse engineering of change decisions can be done in situations where a perfect result is not required. In cases where a near-perfect result is enough, we do, however, believe that reverse engineering change decisions is worthwhile. This line of thought is supported by our results, which, in summary, show that we are able to reverse engineer the most major trends with very high accuracy, but that certain nuances of the changes can be misinterpreted.

## 5.1 Implications for Research

The implications of our findings should encourage research to further study APIs by relating them to existing software evolution theories. Considering that we were able to successfully reverse engineer the changes made to the API in this study, it opens up possibilities to base future research on, for example, the types of software evolution and software maintenance. Since the kappa analysis showed that there was a high level of agreement between our classification and the actual motives to the change, it shows that the types of software evolution and software maintenance could even be used as the basis for a large-scale quantitative analysis to map the types of changes made to APIs.

By analysing the API based on Lehman's laws, we were able to find that the law of Conservation of Organisational Stability had little impact upon the API investigated in this study. Our results show that the opposite of what the law suggests, in its current form, is true for this API. However, since only a single API was studied, we cannot generalise this finding for all APIs. Research should therefore be wary of assuming that this law is true for APIs and future work could be undertaken to fully determine its validity in regards to APIs.

## 5.2 Implications for Company A

The changes introduced to the API are mainly based on the expertise and experiences of the API architects at company A. There has not been a scientific approach to maintaining the system and the knowledge from research has not directly affected the way that the API's evolution has been managed. With our work, we hope to provide further insight into how the API is maintained. Our analysis of the API based on Lehman's laws should further assure that company A is moving the evolution of the API in the right direction. Our research proposes a dilemma for company A in how to tackle challenges related to the law of Conservation of Organisational Stability. Determining if the benefits of continuously and frequently updating the API outweigh the side-effects that these updates could have on the application developers, is something that company A will need to consider.

One way of finding the solution to this problem would be by in a more structured way including the application developers in the feedback loop. Since the application developers are on the receiving end of the changes, we believe that their needs have to be assessed before the problem can be solved. With our study we identified both that the law of Feedback System is of great importance to company A and that it is one of the key areas of API development that company A wishes to improve. One of the challenges posed by software evolution is that the activities carried out are often intangible and non-objective [7, 19]. Based on this, we propose that our study which has granulated the activities is used as a basis for communicating the current stage of the evolution of their API. We hope that this classification can be used in its current form, to help quantify the activities and to provide an overview of them. We also hope that the approach we used can be used to predict which types of activities should be emphasised in the future evolution of the API. With our study we have also demonstrated that the classification of changes is achievable, which, in turn, should encourage company A to adopt this method of classifying change.

# 6. FUTURE WORK

With this study, we conclude that studying API evolution with the help of existing theories, provides a new and structured approach to the field of API evolution. Our study shows that functionality is the largest driving force of change in the evolution of the studied API. It also suggests that changes to APIs occur infrequently and sporadically. This indicates that the law of Conservation of Organisational Stability did not play a significant role in the evolution of this API. However, given that this study was only performed on a single API, we cannot generalise this finding. Furthermore, we have identified the importance of having a structured feedback loop when maintaining an API. We have also provided company A with a foundation for facilitating such a change.

To build on our study, we propose that a study on the eco-system's role in the evolution of APIs is conducted. We believe that this area could be a major source of improvement for API evolution. Exploring the benefits of including application developers and additional entities of the eco-system in the feedback loop, is a good starting point for company A. In such a study, we would suggest that the types of soft-

ware evolution and software maintenance [7] could be used as a mean of facilitating discussion between the entities. It could also be possible to study the effect that changes to APIs have on the application code, in such a study.

To further investigate what drives API evolution, we believe that API evolution should be studied from a business point of view. We therefore suggest that a similar study is conducted, but with the goal of analysing the change of an API based on business motives. Positioning a study around business motives would help with identifying the changes before they are formulated into technical changes, and it would provide a new and fresh angle to the field of API evolution.

Lastly, based on our successful attempt to reverse engineer the changes made to this API, we suggest that a quantitative study is conducted to investigate the distribution of the types of software evolution and software maintenance and the applicability of Lehman's laws on a large scale.

# 7. REFERENCES


[1] Systems and Software Engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[2] L. M. Afonso, R. F. d. G. Cerqueira, and C. S. de Souza. Evaluating Application Programming Interfaces as Communication Artefacts. *System*, 100:8–31, 2012.

[3] C. Andersson and P. Runeson. A Spiral Process Model for Case Studies on Software Quality Monitoring - Method and Metrics. *Software Process: Improvement and Practice*, 12(2):125–140, 2007.

[4] I. Benbasat, D. K. Goldstein, and M. Mead. The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, pages 369–386, 1987.

[5] J. Bloch. How to Design a Good API and Why it Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507. ACM, 2006.

[6] J. Bloch. *Effective Java*. Pearson Education, India, 2008.

[7] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[8] S. Clarke. Measuring API Usability. *Doctor Dobbs Journal*, 29(5):S1–S5, 2004.

[9] J. Cohen. Weighted kappa: Nominal Scale Agreement Provision for Scaled Disagreement or Partial Credit. *Psychological bulletin*, 70(4):213, 1968.

[10] D. Dig and R. Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398. IEEE, 2005.

[11] D. Dig and R. Johnson. How do APIs Evolve? a Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[12] Git. http://git-scm.com/. Accessed: 2015-05-08.

[13] J. Henkel and A. Diwan. Catchup!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283. ACM, 2005.

[14] M. Henning. API Design Matters. *Queue*, 5(4):24–36, 2007.

[15] D. Hou and X. Yao. Exploring the Intent Behind API Evolution: A Case Study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, pages 131–140. IEEE, 2011.

[16] M. M. Lehman. On Understanding Laws, Evolution, and Conservation in the Large-program Life Cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[17] M. M. Lehman. Programs, Life cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[18] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[19] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[20] T. McDonnell, B. Ray, and M. Kim. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[21] D. G. Messerschmitt and C. Szyperski. Software Ecosystem: Understanding an Indispensable Technology and Industry. *MIT Press Books*, 1, 2005.

[22] M. Piccioni, C. A. Furia, and B. Meyer. An Empirical Study of API Usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 5–14. IEEE, 2013.

[23] R. Robbes, M. Lungu, and D. Röthlisberger. How do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[24] C. Robson. *Real World Research*, volume 2. Blackwell Publishers, Oxford, 2002.

[25] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[26] C. B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[27] L. Shi, H. Zhong, T. Xie, and M. Li. An Empirical Study on Evolution of API Documentation. In *Fundamental Approaches to Software Engineering*, pages 416–431. Springer, 2011.

[28] J. Stylos, S. Clarke, and B. Myers. Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. In *Proceedings of the 18th PPIG Workshop*, 2006.

[29] Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *2013 IEEE International Conference on Software Maintenance*, pages 458–468. IEEE, 2006.

[30] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, 2013.