Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Scenario-based Modifiability Evaluation of Service-Based Systems: Tool Support for Lightweight Scenario Templates

Gerhard Breul

Course of Study:	Informatik, B.Sc
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Justus Bogner, M.Sc.
Commenced:	June 3, 2018
Completed:	Decenber 3, 2018

Abstract

In modern software development, modifiability has arguably become one of the most important software quality attributes. There has been extensive research on the topic of the evaluation of a system's or architecture's ability to be modified. Scenario-based methods such as ALMA or SAAM have been around for a long time, and have proven to be effective ways to evaluate software. However, these methods are usually quite expensive in terms of time and require a big part of a project's stakeholders to align. One thing there has been a notable lack of is the modifiability evaluation specifically of systems using a service-based architecture. Properties of this architectural style can be useful to make more efficient assessments of a software's modifiability.

This work seeks to propose a method to evaluate modifiability of service-based systems using a scenario-based analysis approach. Special emphasis is placed on this method being lightweight, i.e. less time-consuming and more flexible than similar methods. To achieve this, existing works on the topics of scenario-based software evaluation and the evolution on service-based systems were studied. Based on the acquired knowledge, a model of software change and an approach to analyze a system's modifiability utilizing this model was designed. A tool was created to support our proposed method, testing and application of which in turn lead to iterative adjustments to the method. To demonstrate the insights that can be won using this method as well as how it is used, we perform an example application of it.

The result of this work is a method that exchanges some of the reliability and accuracy of other scenario-based methods for flexibility and brevity. While the method achieves its goal of being lightweight, real-world testing and validation may lead to improvements to it.

Contents

1

1 Introduction

Before getting into the work itself, we will describe its relevance, scope, and methodology in this chapter.

1.1 Motivation

As the rate at which new technologies emerge increased over the last decades, the need to be able to adapt software to utilize them did so as well.

This means that today's software needs to be able to be deployed, updated and expanded fairly quickly and efficiently.

Because of this, for modern software systems, most development efforts (and therefore costs) tends to no longer be found in the initial pre-release stage, but afterwards, in the maintenance stage. To facilitate both a timely release as well as a higher degree of flexibility for later improvements.

This constitutes a shift in paradigm from the early days of software development: In the seventies, the cost of development was usually front-loaded (maintenance made up about 50% of the total cost incurred). By the end of the 90s, the maintenance-to-total-cost ratio had increased drastically to typically over 80% [PPR03].

Naturally, a development cycle focused on continuous maintenance leads to rising maintenance cost, which in turn increases the need to be able to estimate how well an architecture would react to being modified in certain ways, such as the addition of new features or functionality. The quantitative approach to make this kind of prediction would be to look at certain metrics of the system for evaluation. Given the proper tools and support structure, this has the advantage of being easier to perform than other methods; more often than not, the information required can be raised automatically and needs comparatively little in the way of actual human input. However, metric-based software modifiability evaluation has its limits: Many of the metrics people like to use are aged and often have an unclear impact on maintainability[OW14]. A very basic example would be the „Lines of Code"-metric: One would assume this metric to generally be negatively correlated with modifiability. If you, however, compare two semantically identical software applications, one of which has certain values hard-coded into it and the other one stores them in a configuration file (creating considerable overhead in code length), this assumption shows itself to be inaccurate at best. Here, the application with more lines of code is the one with considerably better modifiability. Another example is Cyclomatic Complexity, which measures the number of independent execution paths. While it does sometimes correlate with human-perceived complexity (and thus, maintainability), other factors play a bigger role in making code hard to understand for maintainers. A set of metrics better suited to the task of maintainability evaluation has been suggested by [OW14].

Another problem of building a software system to fulfill a certain metric is that developers might be tempted to build their software to satisfy it instead of its underlying quality requirement.

An alternative way of analyzing software quality attributes in general and the attribute or modifiability in particular are scenario-based approaches. They are a lot more labor-intensive, as the actual evaluation must be done manually by humans instead of being automated, but result in a deep understanding of the architecture (or system) under consideration and its strengths and weaknesses [Woo12]. The general idea is to identify situations the system is likely to encounter or that are critical in some way and analyze how the system would behave in these kinds of situations. We call these situations 'Scenarios'. Scenario-based methods are usually quite expensive, as they need a considerable portion of the people involved in a project to participate and align their schedules. There are many well-established and tested methods of this kind for assessing quality attributes such as modifiability of an architecture, some of which we will deal with more closely later.

Another consequence of short development cycles is an emphasis on the utilization of practices and architectural styles that try to make it easier for the maintainer of a software to adapt it to new requirements. One such architectural style are service oriented architectures citebieberstein2006service.

Software systems utilizing this type of architecture have some properties that positively affect modifiability, such as their strict modularity and well-defined interfaces. These properties do not only make it easier to apply changes to a system, but can also be used to simplify modifiability analysis due to some of the restrictions and rules they bring about. Although there is ample research on the topic of both methods for evaluating modifiability and service-based systems and their maintenance, to the best of my knowledge, there is none regarding the scenario-based modifiability analysis of service-based systems in particular. An evaluation method specialized for service-based architectures might avoid the drawbacks of conventional techniques such as being very expensive and time-consuming.

1.2 Scope and Research Objectives

The goal of this work is to create a scenario-based modifiability evaluation method which is more lightweight than many of the established ones, at the cost of only being applicable to software systems based on a service-based architecture. Therefore, the first question to answer is whether there are properties of service oriented architecture (SOA) that can be helpful in determining a system's reaction to being changed, and if so, how they can be used.

A meta-model for change in a SOA, and based on it, a process by which its instantiated models can be evaluated and quantified in some way to represent the modifiability of the modeled system needs to be developed. To create a model, we first need to be able to categorize software changes in a meaningful way. To create a useful process, several questions have to get answered, such as which level of abstraction should be used: a too fine-grained view might improve the results' reliability, but can also make the method more complex and unwieldy, whereas a high abstraction level might satisfy the requirement to be easy to use, but forfeit some of the analysis' accuracy.

Special emphasis is put on the word “lightweight”. Time and effort put into the analysis should be kept as little as possible. Because of this, we need to find out which information is needed to create a viable evaluation of a system’s modifiability, and which information does not have a big enough impact to justify the effort associated with its inclusion.

1.3 Research Method

To create this new method of modifiability analysis, the first step was to study a number of publications on the relevant topics: Scenario-based software analysis methods and the modification of service-based systems.

Next, the gathered knowledge was used to create a meta-model for changes to a service-based software system. To achieve this, models used in existing scenario-based evaluation approaches where used as a starting point for our own model. As a priority of this work lies on creating a lightweight method, alterations to the initial model and method were made with the goal of simplifying its instantiation and improving ease-of-use while maintaining as much accuracy as possible. To this end, some trade-offs had to be made. For example, the benefits of using certain points of data (such as metrics) were weighted against the cost of raising and analyzing them.

After a model and a method to analyze it was developed based on the previously researched methods, the next step was to build a tool to assist in applying it. In doing so, more possible trade-offs became apparent. Because emphasis was once again put on being intuitive and easy to use, the model was adapted to accommodate this change wherever it seemed beneficial.

2 Technical Background

This chapter is concerned with the technical knowledge that is necessary to create the modifiability analysis method we want to develop. Firstly, we take a look at service-based architecture. The second part deals with the maintenance and evolution of software in general. Lastly, we discuss some of the scenario-based methods that can be used to evaluate the modifiability of a system.

2.1 Service-Based Systems

The term "service-based architecture" describes the architectural style of a system made up of several interconnected services. Such a system is called a service-based system. Although there is no universally accepted exact definition of what makes a service-based system, there seems to be a general agreement of what the expression means. OASIS calls it "A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectation" [OAS06]. Below, some attributes such a system is usually associated with are listed. For the purpose of this work, these points will also serve us as a definition:

- Modularity: a software application is divided into services, each of which performs lower-level actions. These services are (combined in a composition/composed) to accomplish more complex tasks.
- Independence: Services are loosely coupled and communicate only via certain (usually network) interfaces in previously agreed upon ways, so-called 'Service Contracts'. This allows the system do be distributed across multiple machines.
- Black Box: For other services as well as the user, a service's inner workings remain unknown. Form the outside, only its interfaces and operations are visible. This also means that services within a system can be owned by different entities.

Another helpful piece of information when it comes to understanding what service-based systems are is the SOA Manifesto [Gro+09]. It states some general priorities and values shared by this architectural style. Some of the more important ones for the topic are as follows: For example, the authors judge evolutionary refinement to be more valuable than pursuit of initial perfection. This goes along with the previously mentioned shift in design philosophy: A system does not need to be perfect on release; It should, however, be maintained to keep up with changing requirements. Other suggested priorities are "Shared services over specific-purpose implementations" and "Flexibility over optimization"; Services should be built as multi-purpose tools instead of solving a specific problem. Such priorities attempt to play to the strengths of SOAs, such as the reusability of a service in a different system.

In a service-based system, communication between services is defined by service contracts [Pap08a], which specify functional requirements (such as how to invoke a service or what results to expect), non-functional requirements (such as quality attributes of the service), and rules of engagement.

An architecture that specialized for modularity comes with its own set of advantages and drawbacks: One of the biggest advantages being, as previously mentioned, improved maintainability [WGEZ16]. Changes to a single service often remain contained to it and therefore rarely affect others in a meaningful way. These changes are often called shallow changes [Pap08b] and are usually comparatively easy to apply, due to the relatively small size of the affected code.

Bigger changes, for example those that have an effect on the relationships between services or those that affect multiple services at once, are often called deep changes [Pap08b]. With these changes, altering one component can result in having to alter another. Changes may ripple to different parts of the system. Because of this, they require more planning and are much more difficult to deal with than shallow changes. This is true for SOAs as well as other architectures (although you would likely replace the word ‘service’ with ‘component’ for different architectures). Predicting ripples is a mayor difficulty for most architectures when predicting the impact of a certain change. Maintainers often heavily underestimate the amount of components influenced by a deep change: Empirical results suggest that software engineers predict only about 50% of all necessary changes when analyzing the impact of a change (Lindvall & Sandahl, 1998)[LS98]. Service-based systems can alleviate this problem to a degree, as services communicate only in very specific ways over certain interfaces. These restrictions limit the amount of possible ripples of a change to a certain service to itself, its clients (services that use operations of the original service) and its providers (services of which the original service is a client). Some research has been done on how to systematically determine which of these services are actually impacted by a change in [WYZ10], xiao2007supporting.

Other advantages of SOA that play a role for modifiability analysis include the reusability of services: As stated in [Gro+09], shared services are given a higher priority than specific-purpose implementations, meaning a service has more possible applications than it would have otherwise. This makes it more likely that an architect can reuse services already at hand when building new systems. This way, some development time can be saved.

SOA does, however, bring some drawbacks with it. For example, having parts of your system only communicate through network interfaces means that it will have to deal problems such as latency or unavailable services. Under these circumstances, failure of parts of the system can rarely be completely avoided. For those situations, a system should be robust enough to handle services responding late or failing gracefully, especially if a high degree of availability is required [Tre15]. To achieve this, failure must be anticipated prior to the problem arising. This contributes to higher complexity in service-based systems. A related weakness of SOA is its increased overhead; given the same functionality, a monolithic application will most likely work faster and more reliable an equivalent distributed system (so long as scalability does not become a relevant factor, at which point, distributed systems have an advantage again).

Whether using SOA for a system is advantageous is therefore dependent on its requirements: If it needs the ability to adapt quickly and often, be expandable and scalable and a very fast reaction time is not crucial (such as in a web application), such an architecture can be a viable choice.

On the other side, if a system is required to have very low latency, does not need the evolution advantages or scalability offered by such modular systems or are simply too small to justify the increased overhead and complexity, a service-based system will likely be outperformed by other architectural approaches.

One prominent example of SOA are web services. Even though there are other systems that fit the definitions, web services are often synonymous to service oriented systems.

An extreme form of SOA can be found in Microservices: Here, the size of individual services is kept very small, as the name suggests. A single service shouldn't be bigger than what can be managed by a small team [New15]. This naturally amplifies the advantages and disadvantages of service-based systems even more.

2.2 Software Evolution and Maintainability

Unsurprisingly, the question of how to best maintain software has been on the mind of many researchers.

Swanson [Swa76] identifies three types of maintenance based on the intention behind it:

- perfective maintenance: perfecting a system's performance, processing efficiency, or maintainability.
- adaptive maintenance: adapting a system to changes in its environment.
- corrective maintenance: correcting a system's processing, performance, or implementation failures.

These three categories have been the foundation of many consequent attempts at classifying different types of maintenance by other researchers; However, those usually alter the original definitions, such as the IEEE Standard for Software Maintenance [98], and often add additional types to better suit their specific needs. Chapin et al. [CHK+01] introduced a classification scheme that no longer relies on the intention behind a change to classify it, as such intentions can be hard to reconstruct from the maintenance work itself without consulting the people involved and may change depending on the organizational and political context. Instead, they base their proposed classification on work done to the software, code, and the customer-experienced functionality. Their classification distinguishes between twelve types divided into the categories 'Support Interface', 'Documentation', 'Business Rules', and 'Software Properties'.

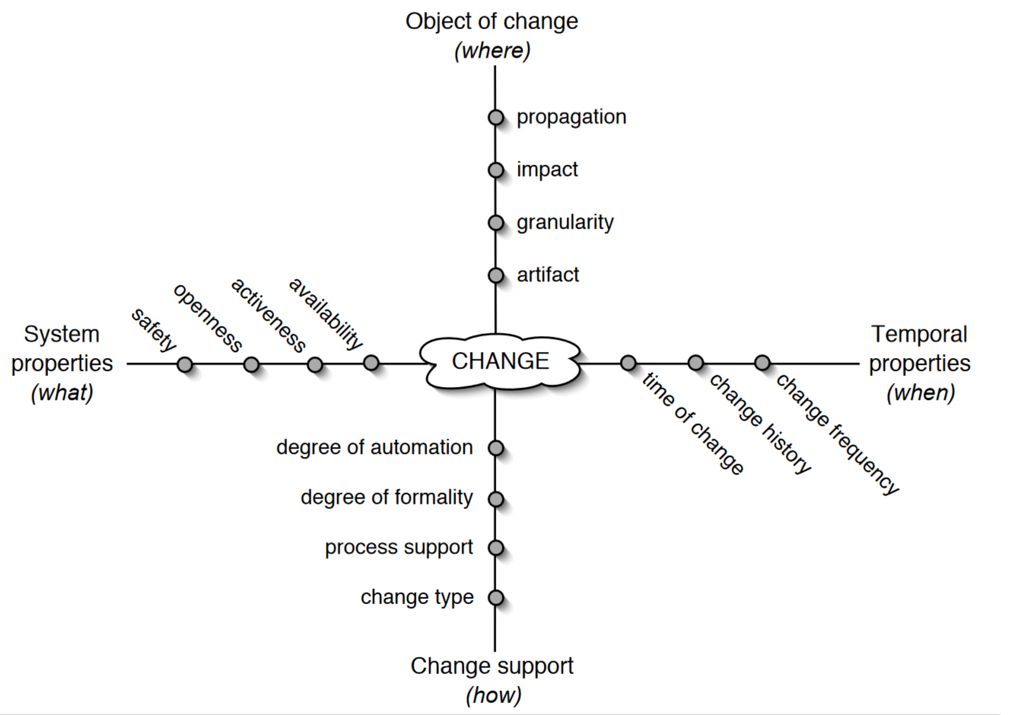

Figure 2.1: Taxonomy of Software Change

Buckley et al. [MBZR03] deal with a “taxonomy” of software change, classifying it with the help of a total of 15 dimensions that each describe either the where (object of change), what (system properties), when (temporal properties) or how (change support) of a certain alteration (figure 2.1). These dimensions are inferred from system properties, its supporting structure, its history and the change itself. Applied to service-based systems, some of these dimensions are fairly simple to determine. For example, the ‘Openness’ variable, a part of the system properties that indicates whether a system is built to allow for extensions, can generally take two values: Open and closed. For service-based systems, this dimension’s value is effectively always ‘open’; Therefore, we do not need to consider both possibilities, simplifying the process of classification (and by extension that of evaluation).

Another helpful concept when considering evolution specifically of services is the division of changes into “deep” and “shallow”, as introduced in [Pap08b]:

- Shallow changes: Where the change effects are localized to a service or are strictly restricted to the clients of that service.
- Deep changes: These are cascading types of changes which extend beyond the clients of a service possibly to entire value-chain, i.e., clients of these service clients such as outsourcers or suppliers.

To deal with shallow changes, a system merely needs an appropriate versioning strategy, whereas to accommodate deep changes, a change-oriented service cycle life is required, meaning it needs a foundation to gracefully redesign services in a manner that prevents it from becoming convoluted.

In terms of Buckley's taxonomy of change, the main difference of shallow and deep changes are found in the dimensions which answer the question of 'where', those being Artifact, Granularity, Impact, and Change Propagation. Deep changes have a very coarse granularity, high level artifacts, and an impact that is at least close to system-wide due to strong change propagation. Shallow changes, on the other hand, have at most a service-wide granularity and do not propagate past the original service's clients. One important prerequisite for correctly classifying a change is therefore a method to accurately predict change impact. Change impact analysis tries to find out to what extend a change propagates to other parts of a system [Arn96]. However, as previously mentioned, in practice, it's a very challenging task and will often yield inaccurate results. To perform change impact analysis, one requirement is to find out the dependencies between the different components. Effectively, we want to know the possible places a change could theoretically ripple to. The next requirement is to answer the question of which of these components a change will actually impact. For big monolithic systems, finding the solution to these two problems can prove to be quite difficult: Dependencies between different classes might not be easily visible from the architecture without analyzing the implementation. the tightly coupled nature of such systems and their size can result in a high quantity of interconnections. Judging which of those dependencies a given change will propagate through is a challenging task for maintainers. For service-based systems, these problems still exist, but loose coupling and communication restricted to specific interfaces and operations between services alleviate much of the difficulty introduced by a heavily interconnected architecture. This, of course, does not take changes to a service itself into account: Theoretically, the difficulties pertaining to a monolithic system can be found in single services as well, as every service can be such a system in itself. However, for our purposes, we will disregard those problems due to single services in reality being small enough to be maintained without complicated propagation analysis methods.

One model and process to determine change propagation for a program with known dependencies is presented in [Raj97]. Here, dependencies are represented as edges and components are represented as nodes in a graph. Impact analysis is done by marking each dependency of a changed component as an inconsistency, and judging for each inconsistency whether the dependent component needs a change itself. While this method does not help us apart from analyzing impact within a service, a similar approach could be adapted for an inter-service level. A thorough approach tailored to SOA impact analysis is provided by [WYZ10]. This method takes two layers into account, the process layer and the service layer. A classification of service change and mapping to one of ten proposed impact patterns is performed in order to predict change propagation.

2.3 Scenario-Based Evaluation Methods

The basic principal of scenario-based software evaluation is to first analyze the software, its environment, and requirements to find scenarios that are likely or critical in some way to one or more of its quality attributes. Next, it is systematically determined how the software reacts to these scenarios. These reactions are then evaluated with regard to (a) specified quality attribute(s). Such methods are called qualitative approaches, as opposed to metrics-based evaluation or quantitative approaches. According to [Woo12], most architectural assessment methods defined in the research community are scenario-based, with a consensus having been reached that this technique should be the basis of any effective evaluation technique.

In this part, we will familiarize ourselves with scenario-based approaches. Three of the most common methods are SAAM, ATAM, and ALMA.

The Software Architecture Analysis Method (SAAM) [KBAW94] is the oldest method of this type. Its main concern is to determine how well a certain architecture performs with regard to one or a certain set of quality attributes. Applying the same set of attributes and scenarios to two different architectures allows for a comparison between them, which SAAM was originally developed for [KABC96]. While it was designed for modifiability, it proves useful for other quality attributes as well. The original approach was iterated on by the authors over time. A (modern) SAAM evaluation is done in six steps:

1. Describe architecture: A system is described by its components and the relationships between them. The original approach also suggested the use of simple language to be able to consistently describe different architectures, and therefore forcing a common level of understanding.
2. Elicit scenarios: Developing scenarios means covering all anticipated changes and activities the system needs to support. Because scenarios from all different stakeholders' perspectives are needed, SAAM requires a big part of the people involved to take part in this activity.
3. Scenario classification and prioritization: Scenarios are classified as either direct scenarios, which are applied to the system as-is, as or indirect ones, which require the system to change. Prioritization means that all scenarios are given a priority according to the importance they have for the evaluation goal. Usually, not all scenarios will be considered further.
4. Perform scenario evaluation: In this step, scenarios are applied to the candidate system. For indirect scenarios, this means that each change to or addition of a component or connection between components is listed.
5. Reveal scenario interaction: A scenario interaction means that multiple (indirect) scenarios necessitate change to the same component or connection. This can point to a problem or weakness in the architecture.
6. Reveal scenario interaction: A scenario interaction means that multiple (indirect) scenarios necessitate change to the same component or connection. This can point to a problem or weakness in the architecture.
7. Overall evaluation: In the last step, each scenario is given a weight according to its importance relative to the other scenarios. The authors suggest to once again assemble all stakeholders of the project for accurate results.

For other approaches, we will not go into detail about how each one is performed step-by-step in this chapter. Instead, we will focus on similarities and differences between them.

The Architecture Tradeoff Analysis Method (ATAM) [KKB+98] aims at finding tradeoffs, sensitivity points, and risks in a given architecture. As such, it is most effective when used early on, although it can be used at any stage in the software life cycle.

ATAM is divided into two phases: The first is concerned with describing and analyzing the architecture, its quality attributes, possible risks, and trade-offs, for example by generating a quality attribute utility tree. Phase two consults a wider range of stakeholders to prioritize, evaluate, and elicit more scenarios. The result is a deeper understanding of the strengths and weaknesses of the analyzed architecture in the form of tradeoff- and sensitivity points. Decisions about these should be carefully considered. Notably, ATAM does not emphasize in-depth assessment of one quality attribute in particular, but tradeoffs between them.

Architectural Layer Modifiability Analysis (ALMA) [BLBV04] is, as the name suggests, an approach specifically designed for modifiability analysis. It pursues one of three goals, namely maintenance cost prediction, risk assessment, and candidate architecture comparison. Each of these goals slightly modifies the approach. Therefore, the first step in performing it is to decide on one. From this point on, the method bears some resemblance to SAAM, as an architecture description followed by scenario elicitation and evaluation is performed and the evaluation concludes with an interpretation of its results. A major difference lies in the techniques specified for each step by each method: For example, ALMA mandates a thorough approach to scenario elicitation while SAAM does not explicitly mandate any technique. Not having fixed methods for scenario elicitation, scenario analysis and evaluation has the benefit of flexibility, but may compromise repeatability of the method. A closer investigation of ALMA can be found in the next chapter.

A comparison between different scenario-based evaluation methods can be found in [BG04]. In this work, the authors compare the three methods named above as well as the Performance Assessment of Software Architecture (PASA) [LC02], an approach aimed at performance analysis. Among the findings of this comparison is an overlap in some of the activities performed in each of these methods, which results in a sort of generic approach to scenario-based evaluation. In essence, the authors identify a Planning and Preparation stage, an Architecture Description stage, a Scenario Elicitation stage, an architecture analysis stage and a final stage in which results are interpreted and presented.

3 Related Work

While there is a large body of research dedicated to both scenario-based software analysis and evolution of service-based architectures, we are unaware of any work done on the intersection of these two fields. In this chapter, a selection of literature that was especially relevant or influential to this work will be discussed.

Probably the biggest inspiration for this work is the already mentioned ALMA method [BLBV04]. It holds special relevance to this work: As a scenario-based modifiability analysis method that can plausibly be applied to a service-based system, it appears to be a good candidate to act as a basis for the method we are designing. ALMA has five steps:

1. Setting a goal: As mentioned previously, ALMA starts out by determining whether the goal of the analysis is cost prediction, risk assessment, or candidate architecture comparison. The choice of goal will impact scenario elicitation and evaluation as well as the interpretation of the result.
2. Describing the architecture: The information raised in this step is primarily required for impact analysis in the scenario evaluation step. As such, the architecture description needs to contain the decomposition of the system into its components as well as their relationships to each other and the environment.
3. Eliciting scenarios: For any scenario-based approach to software evaluation, the quality of the scenarios used is of paramount importance. For this reason, the authors propose the use of classification of scenarios into equivalence classes to decrease the number of scenarios that have to be considered and the use of change categories to focus on relevant scenarios. This method can be applied top-down, going from a classification to a specific scenario, or bottom up, starting from a scenario and classifying it. Which change categories are relevant depends on the goal chosen for the evaluation: For cost prediction, scenarios should describe situations that will occur with a high probability, for risk assessment, scenarios should highlight difficulties, and for architecture comparison, scenarios should focus on differences in either risk or maintenance effort.
4. Evaluating scenarios: ALMA evaluates scenarios by identifying affected components, determining the scenario's effect on them, and identifying ripples. The last step can recursively lead to having to determine the scenario's effect on more components and more ripples. When the effect on all components is determined, the scenario's impact needs to be expressed according to the goal chosen.
5. Interpreting results: Finally, a conclusion is made, taking all scenario evaluations into account.

While ALMA cannot be considered a very lightweight method, it can be performed relatively quickly compared to other scenario-based approaches such as SAAM or ATAM. This is due to its focus on a specific quality attribute and its relatively small group of involved parties (while SAAM and ATAM involve all mayor stakeholders, ALMA only requires the architect and development team).

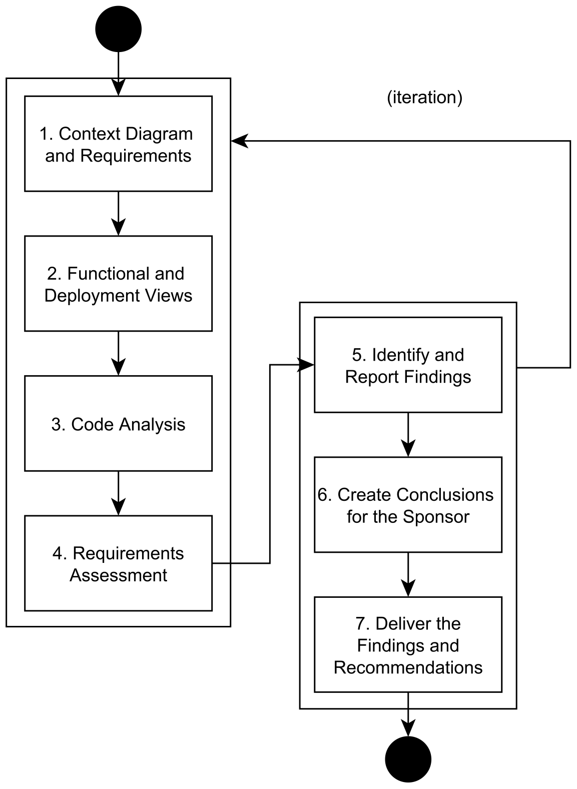
(iteration)
1. Context Diagram and Requirements
2. Functional and Deployment Views
3. Code Analysis
4. Requirements Assessment
5. Identify and Report Findings
6. Create Conclusions for the Sponsor
7. Deliver the Findings and Recommendations

Figure 3.1: Taxonomy of Software Change

A different way of software quality evaluation is proposed by the Tiny Architectural Review Approach (TARA) (Woods, [Woo12]). Its focus, as the name suggests, is on being applicable in situations where other evaluation methods are not feasible due to a shortage of time and/or personnel restrictions. To this end, the TARA method takes a different approach than those introduced up until this point: while the author acknowledges that a consensus among the research community seems to be that "scenarios should underpin any effective evaluation technique", he goes on to state that eliciting valid scenarios is a rather time-consuming activity involving many of the important stakeholders, and as such, would undermine the method's ability to be quick and simple to use. Therefore, instead of relying on scenarios, TARA uses expert judgment and metrics. This also means that TARA is designed to be used post-implementation to allow for automated elicitation of such data as module structure and dependencies as well as metrics. While an implementation is not necessarily required as no specific sub-techniques are mandated by the method, it is still recommended. The reason for not prescribing sub-techniques is to maintain flexibility and to not discourage the user from applying it due to its complexity. The need for an implementation also means that TARA cannot be considered a purely architectural approach.

3.1 depicts the steps involved in performing this approach.

The tradeoff made to achieve this high degree of flexibility and simplicity is that the result is somewhat less reliable and insightful than a more comprehensive technique.

It is interesting to note that while TARA does not use scenarios on the basis that the elicitation of good ones is supposedly not ensured given the small number of participants, ALMA seems to not suffer the same difficulty despite this number being similarly small. This may be due to ALMA focusing on a single software quality attribute and therefore only requiring a relatively narrow field of scenarios.

While TARA is neither specific to modifiability nor a scenario-based approach, its relevance to this work is mainly founded in demonstrating possible ways to reduce the weight of an evaluation method while still delivering a useful set of results.

Arguably one of the most important parts of determining software modifiability especially for service-based architectures is change impact analysis: Often, a difficult part of an evaluation is analyzing which part of what service a change propagates to. An approach to deal with this problem is proposed in [WYZ10]. To perform it, the system in question is first represented in a purpose-built model, the so-called service-oriented business process model. This model consists of two layers, a process layer describing the internal processes and a service layer that contains its supported services. A classification is applied to each change based on its impact on these models. Based on this classification, changes are mapped to one of ten identified change impact patterns. The impact pattern dictates the reaction to the change, including its ripple effects, which in turn cause more changes.

This approach represents a very thorough and intricate method of change impact analysis. While predicting change propagation in this method likely yields accurate results, the information required and complexity inherent to it go far beyond what could be considered lightweight. As such, directly employing this technique for change impact analysis in our method is not recommended.

$$\begin{tikzcd}
p \arrow[r, "dep"] \arrow[d, "\Delta"'] & u \arrow[d, "t"] \\
p' \arrow[r, "dep'"'] & u'
\end{tikzcd}$$

Figure 3.2: Service Evolution Model

The last piece of influential research to be mentioned in this chapter deals with service evolution patterns [WHHC14]. In their work, Wang et al. introduce a service evolution model 3.2. In it, a set of changes done to a service p results in a transition set t in each of its dependencies u. For each dependency with a non-empty transition set, this can be repeated. A very similar idea can be found in the design of the model we propose. Furthermore, with the help of this model, four distinct service patterns are defined: The Compatibility Pattern for changes that do not propagate to other services, the Transition Pattern for those that do, the Split-Map Pattern for splitting services, and the Merge-Map Pattern for merging them.

4 A Method for Scenario-Based Evaluation of Service-Based Systems

The central part of this work is to develop a scenario-based approach to change evaluation that is lightweight and tailored towards service-based systems. To do so, we will apply the acquired knowledge to design a change model which, when instantiated, can be evaluated to gain insight into the modifiability of the system it represents. Much like TARA, this method aims to be useful in situations where little time and focus can be spared. However, unlike TARA, the method proposed here will use scenarios for evaluation.

4.1 A Model for Software Change

The first step in designing our modifiability evaluation method is to develop a model which we can use our method on. As a requirement for this method is to be lightweight, the model should reflect that too: Instantiation of it should need as little effort as possible, and the information required should not be too difficult to obtain. This part of my work is concerned with the development of a model and the considerations that led to the decisions made.

4.1.1 How Do We Utilize the Properties of Service-Based Systems to Improve Modifiability Analysis?

When it comes to modifiability (as well as its analysis), the most useful property of service-based systems is most likely their loose coupling:

As pointed out in [BLBV04], a major source of uncertainty when performing change impact analysis on a system's architecture is the problem that not all connections between components are visible at the architectural level. An incomplete picture of inter-component dependencies means that if a change propagates through one of these unknown connections, it cannot be predicted. However, in a service-based system, all communications between components (or in this case, services) and therefore possible ripples are limited to specific interfaces and their operations. Assuming service contracts and dependencies are available, many of the connections become visible, which makes it far easier to find out all of the dependencies between services in a system. This, in turn, may let us avoid some of this inaccuracy when assessing change propagation.

A related advantage of loose coupling is its ability to prevent changes to propagate beyond component borders in the first place. Because communication between services in a loosely coupled system occurs only by way of certain, specified interfaces and in predefined ways, a change to one service can propagate to another only if it affects the interface in some manner. For example, a change to the service contract of a REST-web-service must be present to allow for propagation. The thing to find out now is whether or not a change to an interface indeed propagates to the services connected to it. Essentially, this is a question of compatibility as discussed in [ABP12]: a change to a message producer will not require a change to its consumers if the new version of the producer is backward compatible, and a change to a consumer will not require a change to its producer if the new version of the consumer is forward compatible. The reverse is also true, meaning that incompatibility of a new version makes change to connected services necessary. Note that consumer and producer are not necessarily equivalent to client and provider.

Another useful property of service-based systems is that the size of each service is usually small compared to a monolith: When performing scenario-based modifiability analysis, we inevitably reach a point where we have to estimate the likely difficulty, needed effort, or cost of the change. This estimation's quality is a central factor to the quality of the evaluation as a whole and usually depends on the evaluator's experience and the code's complexity. After having identified the services that need changing, this estimation can be performed on a per-service basis. It stands to reason that an evaluator can more accurately gauge the cost or difficulty of applying multiple small changes than that of a single big one, as this effectively parallels a 'divide-and-conquer' approach to the problem.

4.1.2 Choosing a level of granularity

An important choice to be made when it comes to developing a model for modifiability evaluation is how closely we want to look at the systems we analyze. Generally speaking, the more fine-grained the artifacts we look at are, the more reliable our method gets. On the other side, this generates the need for a much bigger amount of information to be raised and processed. This, in turn, drastically increases duration and costs of the evaluation. An extreme case of this would be to consider each modified service as a system of its own and apply to a comprehensive approach such as ATAM to it. In reality, this is unfeasible for our purposes because of the sheer amount of time and effort such a method would consume. As a requirement for our method is to be lightweight, we will refrain from considering each service as a system in itself. This way, the information needed about the system is visible in its architecture. This inter-service granularity also allows us to focus on the strengths of service-based systems as described in 4.1.1.

4.1.3 Deciding on which information is relevant

Based on the requirement of simplicity, the SOA properties we want to capitalize on and the level of granularity, we need the raised data to enable us to:

- understand the services and relationships between them to effectively perform change impact analysis.
- efficiently gauge how much effort the implementation of a given change will need.

The initial model for our scenario-based modifiability analysis was defined as: System S = Services, Scenarios, where 'Services' denotes the set of services belonging to S and 'Scenarios' denotes the set of scenarios we want to apply to S.

A service is defined as:

Service = name, description, language, internal complexity, OwnOperations, ForeignOperations, Changes, System, where ‘name’, ‘description’, and ‘language’ describe the name, the service’s purpose and language it is implemented in, ‘internal complexity’ refers to its perceived complexity (for example on a scale of 1 to 10), ‘OwnOperations’ is the set of operations offered by the service over its interface, ‘ForeignOperations’ is the set of operations from other services used by this one, ‘Changes’ is the set of changes performed on the service across all scenarios, and ‘System’ points to the system this service belongs to.

Scenario = name, description, System, Changes, scenarioEffort, where ‘name’ and ‘description’ name and describe the scenario, ‘System’ points to the parent system, ‘Changes’ is the set of all changes to services of the system, and ‘scenarioEffort’ is the sum of the ‘effort’ attribute of all changes in this scenario.

Change = type, effort, description, Ripples, Operations, Service, Scenario, where ‘type’ ϵ ‘Addition’, ‘Modification’, ‘Deletion’ describes whether a change deletes (as described in ‘towards a taxonomy of software evolution’ [MBZR03]), adds, or modifies the service it belongs to, ‘effort’ is a measure of how costly the change is (in terms of time, money or otherwise), ‘description’ is once again self-explanatory, ‘Ripples’ is the set of services this change propagates to, and ‘Operations’ is the set of operations added, deleted or modified by this change.

Operation = name, contract, Service, ForeignServices, where contract describes the part of the service contract relevant to this operation (such as form and content of the input expected, form and content of the output, protocols used etc.), Service points to the service this operation belongs to, and ForeignServices is the set of services that use this operation.

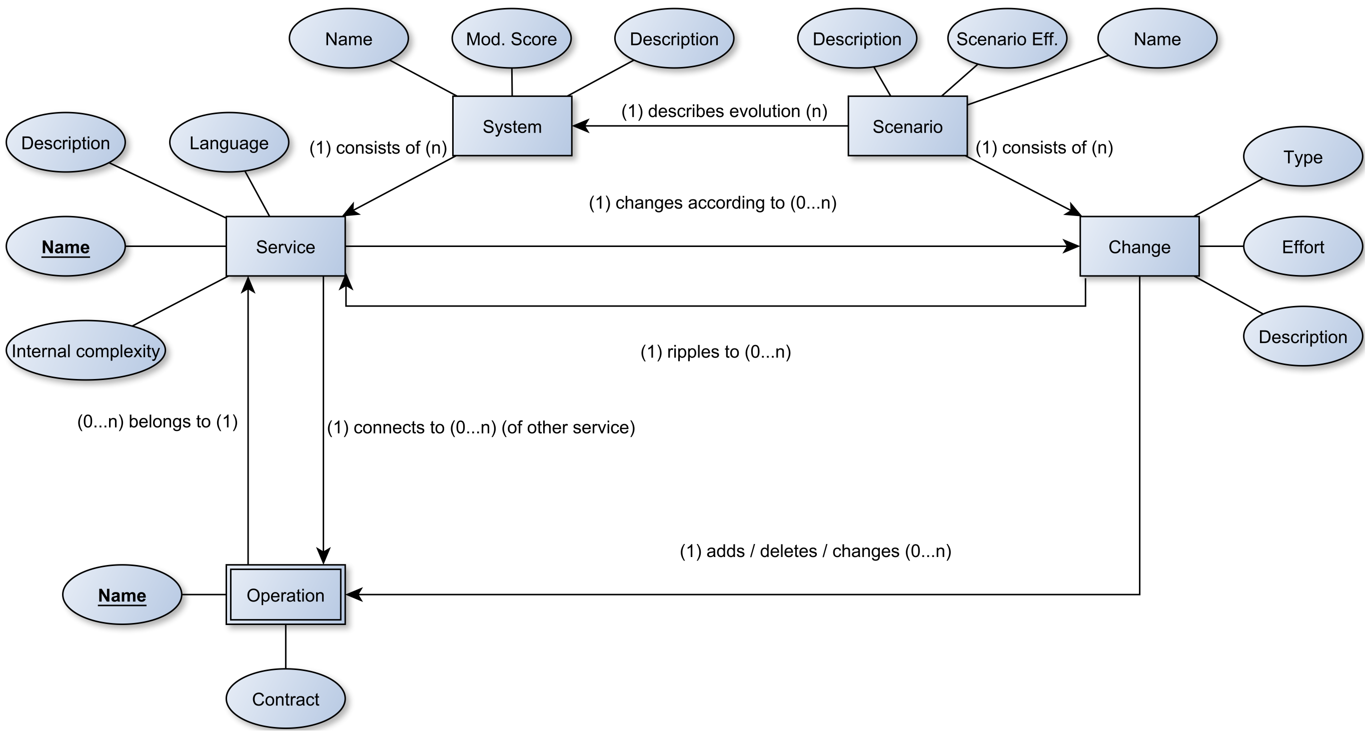

Figure 4.1: Early Version of the Change Model

from here on, to distinguish between the entities of the model and reality, instances of the model will be capitalized

As an early application (without the tool support) of this model to a basic, fictitious web-shop showed, if all needed information can be elicited, it can be used to analyze change propagation by inspecting changed inter-dependencies between Services and Operations and to perform modifiability analysis.

Using this model as a starting point, iterations were made to improve on its design. While this model could be used to achieve its purpose, the effort necessary for creating it sometimes seems to be quite high, considering that this method tries to be as light as possible. For example, it can be tedious to model a service that offers a variety of different operations, as each one needs to be described in detail. This part of the model is meant to facilitate ripple detection, as a formalized dependency graph could be used to partly automate the process. However, this advantage does not seem to outweigh the cost of manually creating this graph. Under the presumption that the system's architect is the one (or one of the ones) performing the evaluation, it is fair to assume that they have a solid grasp on the system's dependencies regardless. Under these circumstances, it may be better to leave the evaluator free reign in choosing the ripple detection method they want to use. In light of its high cost and somewhat limited usefulness, the decision was made to leave out this part of the model.

Another concern was the inclusion of redundant data, namely the language in which a certain component is implemented in and the component's internal complexity: internal complexity was supposed to help the evaluator in the estimation of the effort of a Change. The domain of this variable would be 1-10. While raising this data is not time-consuming, it was pointed out that its purpose was not clear enough and somewhat unintuitive. Users could suspect the value to be used in the evaluation step in some way. Another problem of 'perceived complexity' is its inherent subjectivity: what may seem complex to one person could seem fairly simple to another. Inputting the language of a service was meant to achieve a similar goal: Assuming the evaluator knows which languages the maintenance team is proficient or experienced in, knowledge of the language a certain service is written in certainly improves their ability to gauge how long it would take to implement it. Not raising this information, however, is unlikely to actually impact the outcome: Given an evaluator familiar with the system, it is unlikely for them to not take a service's language into account when estimating the effort of a change. Another reason to not keep 'language' as part of the attributes of Service is that the language is usually a question of implementation, and as such not necessarily visible from the architecture. Applying this method pre-implementation would likely mean for this field to be left blank.

In this final model, most of the non-functional information, meaning information that does not directly influence the evaluation, has been cut. The only instances of it that were kept are the 'description' and 'name' attributes. This resulted in a considerably 'slimmer' version of the model.

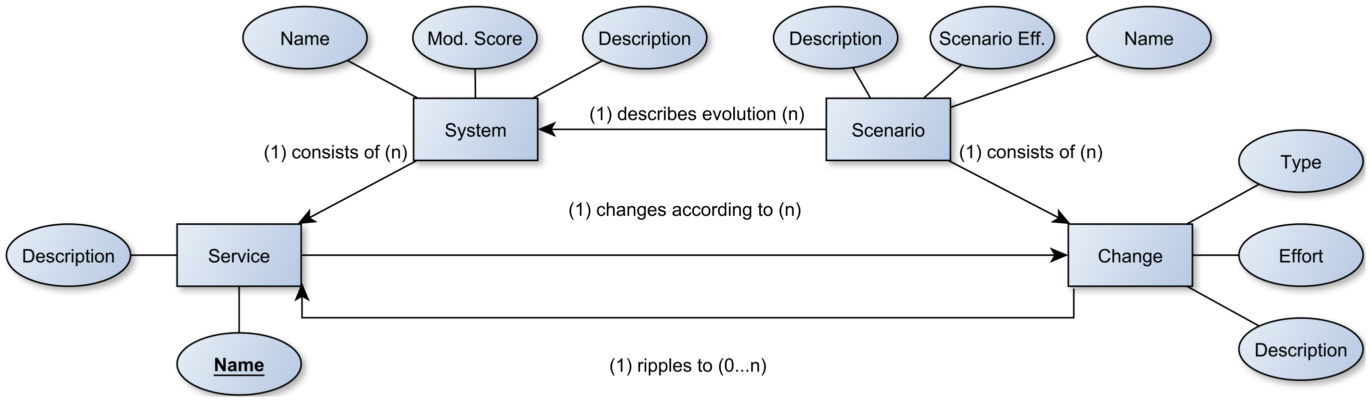

Figure 4.2: Final Version of the Change Model

4.2 A Modifiability Analysis Approach Utilizing the New Model

Now that a model is designed, we can describe an approach to evaluate modifiability with it. This approach consists of the following steps:

1. describe architecture
2. elicit scenarios
3. for each scenario:
    a) determine initial changes
    b) determine ripples
4. evaluate results

These steps do roughly coincide with the generic process of software architecture evaluation found by Babar & Gorton [BG04]. The main difference of this method is our deliberate omission of a dedicated preparation step. This is due to the method being designed to be performed by a single person or a small group, as opposed to more sophisticated approaches which generally have to manage a larger group of people, some of whom may not be very familiar with the method or architecture.

Similar to ALMA, we can use this method to achieve different goals: maintenance cost prediction, risk assessment and architecture comparison. Depending on the goal of the evaluation, some of the steps will be slightly adapted.

In the next part, the execution of each step will be explained in more detail.

4.2.1 Architecture description

The first step taken is feeding our model the data it requires about the system(s) we want to evaluate. For this, we need a description of each service in the system. A dependency graph can be created to aid with ripple detection later on.

When comparing multiple systems to each other, a separate instance of the model is created for each system.

4.2.2 Scenario Elicitation

The process of finding relevant scenarios is central to any scenario-based evaluation method, as an inappropriate set of scenarios will compromise a lot of the evaluation's reliability.

Generally, the process used for scenario elicitation when using this approach is up to the user. This way, a high degree of flexibility is maintained, given that the expected situation in which this method is used is one where time is limited and not many of the people involved are available. If the constrains of the situation allow for it, a structured approach should be considered. For example, ALMAs approach of partitioning scenarios into equivalence classes and change categories can be quite useful for ensuring the quality of scenarios. Another helpful way to find relevant scenarios, especially if time is in short supply, is using general ones (such as the ones proposed by Costa et al. in [CPDM16]) as a starting point to create specific ones.

What kind of scenarios are relevant largely depends on the evaluation's goal: if the intention is to predict future maintenance cost, a good scenario is one that is likely to occur in the future. If the intention is to assess risks or weak points in an architecture, good scenarios are ones that expose those risks, i.e. ones that are difficult to perform. When comparing different systems, good scenarios are the ones that highlight differences between them.

After we found a suitable set of scenarios, we add them to our model. When evaluating multiple systems at the same time, we have the option of assigning a scenario to all systems at once (so each system gets its own instance of the scenario) or only to (a) specific system(s). This can be helpful if we want to, for example, compare different systems with regard to critical components: a scenario that stresses a critical point in one system might not do the same to others. In such a case, comparing systems based on different scenarios can be helpful.

4.2.3 Determining Changes

This step includes two activities: Determining which services are impacted by a scenario and determining how strong these services are impacted. The first one falls in the domain of change propagation analysis, while the second one involves the evaluator estimating the difficulty or effort needed to apply the changes

To find out in which services a scenario will produce changes, we first have to find a set of services that will initially be changed. For each of those, an instance of a Change is created. The 'Ripples' property of this Change can theoretically be acquired using a comprehensive approach to impact analysis such as the one proposed in [WYZ10]. However, for the purposes of our method, this isn't feasible, as formally applying such an approach requires more information and effort than the method itself and would therefore defeat the purpose of a lightweight method.

To maintain as much flexibility as possible, the choice of approach for ripple detection is left to the evaluator. Having a detailed plan of the dependencies within the system (in one's head or otherwise) can be very helpful for this activity.

In our model, changes are classified into three change types: Addition, Modification and Deletion.

Changes that are classified as Modification include any change made to a preexisting service that does not remove it from the architecture. This is the category most changes will fall into.

Changes of the type Deletion remove their corresponding service from the architecture. Adding a Deletion type Change to a Service effectively means that this service will no longer be considered in this scenario. A Change of this type will always ripple to all connected Services.

A Change of type Addition represents a new service being introduced to the system. In our model, this represents a special case. We first need to create a new Service to which we can assign this change. As an exception, this Service is only considered within the scenario where it was created.

Making an accurate assessment of the effort attribute of each Change is a critical part of this evaluation. This attribute will serve as the basis for any further evaluation from this point on. It is raised based on the evaluator's judgment. How it is best represented depends on factors such as the goal of the evaluation as well as preference. A metric we found to be helpful is time: How many work hours or days are required to perform a change or set of changes makes a more objective statement about a system's modifiability. Another useful way is a grading system based on benchmarks, as proposed in SAAM ([KBAW94], [KABC96]) (be it 1-6, A-F, 1-10 or any other ordinal scale). To this end, a well-known and/or likely scenario or change is used to compare other changes against it and based on this, rate them. This metric may not be as objective as 'time needed', as both selecting a benchmark change and comparing and rating changes in relation to each other are fundamentally subjective activities by their nature. However, both approaches deliver values that can be compared, which is an important property for further evaluation, especially if we want to use it to eventually compare systems to each other.

4.2.4 Evaluation

To evaluate the system, we will first consider each scenario by itself. There are multiple different ways to gain insight into modifiability using the given data. Which ones to use is largely determined by what the intention behind the evaluation is. Three ways of eliciting knowledge from our model will be demonstrated here. We call them Scenario Effort, Modifiability Score, and Critical Service.

Scenario Effort: Intuitively, after determining the effort of each Change brought about by a scenario, the next step is to use them to make some kind of statement about the scenario as a whole. For effort based on a time metric, this process is quite straightforward: By summing up all Change efforts of a Scenario, we receive a value that represents the time cost of that scenario. This value will be called 'Scenario Effort'. If Change Effort is represented by a grading system, calculating Scenario Effort becomes less intuitive. One way to approach it is to calculate the average grade of all changes. While this does not take into account the amount of changes and does therefore not represent the actual effort needed for this scenario, it can be used to assess the overall difficulty of implementing a scenario, assuming Change Effort focuses on determining difficulty as well. In this case, however, changing the name form Scenario Effort to Scenario Difficulty might be a more accurate representation.

Modifiability Score: Using the previously calculated Scenario Effort, we are now able to make an assessment on the system as a whole. The approach we chose for this is to calculate a weighted average of all Scenario Effort values. Assigning a weight to scenario is especially useful if our goal is to estimate future maintenance cost: If a certain scenario is three times as likely to occur in the future compared to another one, then its impact on the Modifiability score and therefore its weight should be three times higher as well. If a time metric such as hours/days was chosen as the unit of measurement for Change effort, this represents the average time spent per scenario. In itself, this number does not carry a huge amount of informative value. However, it is useful to compare different candidate systems to each other if each of the systems was evaluated using the same scenarios, especially with regard to the aforementioned goal. Using this method with a grading system, depending on the scenario or change used as a benchmark, the Modifiability Score yields a grade that indicates how difficult evolution of the system is.

Critical Service: An approach more focused on the Services than Scenarios and more tailored towards risk assessment of a single system than comparison or maintenance cost prediction is finding Critical Services. This is similar to (and inspired by) SAAMs "reveal scenario interaction" step: in SAAM, if multiple scenarios effect changes to the same component, this component should be carefully considered when making changes. Similarly, if one service proves to be difficult or laborious to change across multiple scenarios, this service might be of special interest. We calculate the Critical Service by first summing up (if Change effort is on a ratio scale, e.g. time) or averaging (if Change effort is on an ordinal scale, e.g. grade) the Change effort of all Changes that belong to a certain service. Sorting by this value gives us a list of all services ordered by how critical they are. The service(s) topping this list is/are considered Critical Services. While this calculation can be done with an ordinal scale, using a ratio scale offers the added opportunity of calculating how critical a service is by computing the 'effort spent on it to total effort' ratio. In real-world terms, this ratio corresponds to the percentage of maintenance time cost produced by this service.

This list of possible ways to elicit information about modifiability form the proposed model is by no means exhaustive. Other conceivable uses could be for example the detection of critical scenarios, overlapping changes or critical services within a scenario. The usefulness of any information found also depends on the goals pursued by the evaluation.

4.3 Prototypical Software Support

The final part of this work was to create a tool that could be used to perform the proposed modifiability evaluation method. As with the proposed method, special emphasis was put on the 'lightweight' requirement: using the tool to evaluate a system should not demand a lot of time and be intuitive enough to not require a deep amount of knowledge about the method itself.

The tool itself consists of two parts: a RESTful API as backend and a frontend to create and evaluate our model. The API serves to store the instance of the model in a database as it is created and calculate key bits of information such as Service Effort. The frontend consists of a single page application in which the user is first directed to a view showing a list of Systems that are saved in the database (which will be empty at the start) and some information about them, such as the number of services or description. Here, the user has the option of creating a new instance of a System, for which a name and description are requested. Having done this, the user can now populate the new System with its Services. In the service-creation view, the user is once again tasked with providing a name and description for each Service (as well as a pre-filled field indicating parent system). After all Services have been instantiated, the user has the option to proceed to the next step, namely creating Scenarios, or to return to the system-list view to create another System (in which case, the same steps are repeated). Creating a Scenario follows the same pattern as creating a Service: required information includes the name, description and the parent System. For Scenarios, the option is given to create one instance of it for each System instead of specifying one it should apply to. This is a feature added due to the considerations made about comparing multiple candidate systems: for an ‘apples to apples’ comparison, it is helpful to have the same Scenarios as a basis for evaluation. After all Scenarios are described, the user can move on to populate them with Changes. Creating a Change entails selecting the System, Scenario, the affected Service, and a description as well as the Change type, effort and ripples. As mentioned before, the method of determining ripples is up to the user. To generate a complete set of Changes, any Service mentioned in the ‘ripples’ attribute of a Change has to have at least one Change of its own. To prevent the user from trying to evaluate an unfinished model, the API checks for each ripple whether there is a change made to the service it points to. If not, it sends a list of all ‘missing’ ripples. The second important activity in creating a Change is effort estimation. After each Scenario has been populated with changes, the ‘Evaluation’ tab can be used for evaluation. After selecting a System to analyze, relevant information such as the Modifiability Score and Critical Service are presented. A list of Scenarios pertaining to the System is also compiled, where information such as Scenario Effort is visible and adjustments to Scenario weights can be made.

Many of the decisions made concerning the model and method were based on insights gained by testing and applying the tool, such as the omission of the ‘Operation’ entity. Originally, operations were intended to be listed as part of the Service creation process. However, entering every foreign and own operation in Service creation and every added/modified/deleted operation in Change creation turned out to be a daunting and repetitive task. It also contributed to creating a somewhat unintuitive user interface, as the use of this information may not be clear to a user without background knowledge of the model. Two other parts were removed from the model and tool at some point, namely the ‘Language’ and ‘Internal Complexity’ input prompts in the Service creation view. As well as bringing the Service creation view in line with the System and Scenario creation views, this served to make it more intuitive by removing possible sources of confusion (internal complexity was represented on a scale from one to ten) and to slim it down by reducing the amount of data required.

The initial versions of the user interface were iterated on multiple times to support a more intuitive workflow. This lead to a final version that attempts to guide the user through the process wherever possible.

4.4 Use Case Demonstration

To demonstrate the process of the proposed method, we will apply it to an example system. While this does not fully evaluate its effectiveness and efficiency, it should shed some light on how the method can be useful and serve as a candidate version.

The system used for this test represents a web shop and was previously used as an exercise.

It consists of the following services, descriptions of which were found in the documentation:

- *CustomerSrv*: This service provides Create-Read-Update-Delete (CRUD) operations for customer entities as well as an operation to retrieve a refreshed credit rating for a single customer.
- *NotificationSrv*: This service provides operations to send a ”marketing mail”with similar product info to a customer or a ”new product mail”to the sales department. It also provides operations to retrieve the previously sent mails and to add products to as well as retrieve products from its ”new product”database.
- *OrderSrv*: This service provides CRUD operations for order entities including the complicated process to create a new order.
- *ProductSrv*: This service provides CRUD operations for product and product category entities. It also provides operations to get and set the currently available amount of a certain product.
- *WebUI*: The Web UI provides easy read access to the resources of most services.

For Scenario elicitation, we use general modifiability scenarios as described in [CPDM16]:

1. A developer modifies the core logic and internal data sources of a service, but the service contract (uniform interface, supported resource identifications (URI), and representations) remains the same.
2. The representation structure of a resource and its relations to other resources change, and resource identification URI is not affected.
3. The resource representation used in a service is modified and the service correctly processes requests from service consumers that use the old version and consumers that use the new version of the resource under the same URI.

Specific scenarios were derived from the general ones and are described here:

1. After some research, the sales team has decided that this process should now be adjusted and extended. 1: Change the credit rating validation logic. From now on, ratings of 1-4 should be accepted. 2: Change the product availability validation logic. From now on, at least 2 copies of the ordered product have to remain in stock after fulfilling the new order for the product to count as available. 3: Add a new final process step. After creating a new order and before returning the final response, the NotificationSrv should be invoked to send a marketing mail.
2. International shipping will be carried out by a different delivery service than domestic shipping. For each international order, the notification service should send a new type of message to the warehouse operators so they can take the appropriate steps.

3. The ProductSrv has grown over time and is now fairly large compared to the other services. The lead developer has decided to split up the ProductSrv to increase maintainability and scaling efficiency. Two new services will be created: A CategorySrv handling product categories and a WarehouseSrv responsible for product availability. Both have previously been managed by the ProductSrv. A grace period is instituted to give possible unknown service consumers time to adapt in which both the old and new systems are available. Internal clients will be adapted simultaneously to the ProductSrv.

Now that the System is described and Scenarios are elicited, we can proceed to determine the Changes that have to be made for each Scenario:

Scenario 1: In this Scenario, three individual Changes have to be applied, none of which ripple past their original Services.

ID	Service	Type	Effort	Ripples	Description
1	CustomerSrv	mod	0.5h	-	Credit rating check shall no longer return ‘false’ for customers with a credit rating of 4 or better.
2	ProductSrv	mod	0.5h	-	Availability shall return ‘true’ if a product’s available amount after placing an order would be higher than 2.
3	OrderSrv	mod	1h	-	when successfully placing an order, the NotificationSrv should be invoked to send a marketing mail.

It should be noted that Change 3 establishes a dependency between OrderSrv and NotificationSrv. However, this does not cause a ripple because the required functionality (sending a marketing mail) is already established in NotificationSrv.

Scenario 2: Here, new functionality requiring additional information is integrated into the process. This information necessitates Changes beyond the Service in which the functionality is implemented.

ID	Service	Type	Effort	Ripples	Description
1	OrderSrv	mod	1h	CustomerSrv	Add logic to determine whether an order is international. This involves finding out the country of the customer associated with the order.
2	NotificationSrv	mod	3h	-	Add functionality to notify the warehouse team of an international shipment.
3	CustomerSrv	mod	1h	webUI	Add a ‘country’ attribute to the Customer class.
4	WebUI	mod	2h	-	Add ‘country’ to the information required from the customer. For existing customers, a prompt is needed to elicit this information when they make their next order.

This scenario demonstrates how Changes can recursively propagate. While the initial Changes only affected two Services, the ripple from Change 1 caused another ripple, resulting in a total of four affected services. Furthermore, an argument could be made that Change 2 is itself not an intial Change but the result of a ripple from Change 1.

Scenario 3: This scenario assumes the ProductSrv to have third-party consumers. Development of these is not handled by us. Thus, both versions of this part of the system will run in parallel for a time.

ID	Service	Type	Effort	Ripples	Description
1	ProductSrv	mod	2h	OrderSrv, WebUI	With this Change, the functionality to be transferred to the new Services is removed.
2	WarehouseSrv	add	3h	-	New Service responsible for managing Availability
3	CategorySrv	add	3h	-	New Service responsible for managing Categories
4	ProductSrv'	add	1h	-	The launch of the transitional Service. This service is identical to ProductSrv
5	ProductSrv'	del	0,1h	-	Decommission of the old service. This Change will be executed at the end of the grace period.
6	OrderSrv	mod	0.5h	-	Adaption of OrderSrv to the changed endpoints.
7	WebUI	mod	0.5	-	Adaption of the webUI to the changed endpoints.

An interesting aspect of this Scenario is that there are multiple possible options of modeling it. In the option showcased here, the transitional Service ProductSrv' was considered an additional one that had to be added and subsequently removed while the original ProductSrv was adapted to fit the changed requirements. A different approach would have been to discontinue the old ProductSrv and to add ProductSrv', WarehouseSrv, and CategorySrv as its replacement. This would lead do an identical Modifiability Score, but would yield different and arguably worse results for Critical Service, as the effort in splitting up ProductSrv would be partially mapped to ProductSrv'. The third option would be to simply forgo modeling the transitional Service.

Now that all Scenarios are analyzed, the evaluation step can begin. First, the Scenario Effort for each Scenario is calculated as described in 4.2.4. The Scenario Efforts are:

Scenario 1: 0.5h + 0.5h + 1h = 2h

Scenario 2: 1h + 3h + 1h + 2h = 7h

Scenario 3: 2h + 3h + 3h + 1h + 0.1h + 0.5h + 0.5h = 9.1h

Using these Scenario Efforts and weighing them according to their (estimated) probability in relation to each other (Scenario 1: 3, Scenario 2: 2, Scenario 3: 1), a Modifiability Score of 4.85 is calculated. This number represents the average time spent per Scenario in h, assuming the Scenarios are representative for all types of maintenance and weights are chosen according to the ratio of each type's probability.

Another part of evaluation we want to demonstrate here is that of the Critical Service: Adding up the effort of all Changes of a Service regardless of Scenario yields the following results:

CustomerSrv: 1h + 0.5h = 1.5h

NotificationSrv: 3h

OrderSrv: 1h + 1h + 0.5h = 2.5h

ProductSrv: 0.5h + 2h = 2.5h

WebUI: 2h + 0.5h = 2.5h

The Critical Service is therefore NotificationSrv with 3h of effort across all Scenarios due to its high effort in Scenario 2. The OrderSrv should be considered carefully as well, as its total effort is not much smaller than that of the NotificationSrv, and it is targeted by all three Scenarios.

When considering this example execution of the proposed method, one should keep in mind the numbers used are rough estimates and do not necessarily reflect reality. The accuracy of these values strongly depends on the assessor's experience and familiarity with the system.

5 Conclusion

In this work, we proposed a lightweight method for scenario-based software modifiability evaluation specifically tailored to systems using a service-based architecture. We detail the steps taken and decisions made that lead to the creation of this approach. To this end, existing literature on the topics of evolution of service-based systems and scenario-based modifiability analysis was studied. A prototypical tool was designed which can be used to evaluate service-based systems based on the proposed method.

5.1 Discussion and Limitations

In many situations, the designed method trades accuracy and reliability for flexibility and simplicity. An example of such a decision is the omission of the 'Operations'- entity in its model: the main reason to drop it was too much manual labor. One way to fix this problem could be to automate the process based on the implementation. However, this requires an implementation to be available and a tool to detect operations.

Another example is to not mandate a certain scenario elicitation technique. While the technique used by ALMA is not an especially time-consuming one, and a proper scenario elicitation approach should, if the situation allows it, at least be applied informally. doing so may not always be possible.

Due to these decisions, the proposed method may not be well suited for large systems. For those, accuracy would likely suffer without a defined strategy to elicit scenarios and analyze change impact. Mandating certain (heavier) approaches for these activities is not recommended, as this would negate the advantages effectuated by these decisions and may have a deterring effect. While in some situations, specific approaches were proposed, I often refrained from doing so for these reasons. A more structured and in-depth analysis method such as ALMA would be more appropriate in such a situation, if the constraints of the situation allow for it.

The biggest point of contention for this method is its theoretical nature. Without sufficient real-world application and testing, the actual usefulness remains hypothetical. Insights gained from proper testing and evaluation is also likely to lead to more adjustments to the model and process.

Similar to what can be observed with SAAM, repeatability is limited due to the free choice of methods for scenario elicitation and impact analysis. Multiple consecutive applications can yield different results if different or inconsistent methods are used.

Like all scenario-based evaluation methods, the proposed approach relies on the quality of its scenarios to achieve results. If the scenarios do not properly represent situations relevant to the evaluation of whatever software quality (in our case, modifiability) is being analyzed, the reliability of the results is compromised [BLBV04].

5.2 Future Work

As mentioned, at this point, the method has not yet been tested on a real system. Future work applying our method under real-world conditions is therefore required to validate its usefulness.

Furthermore, possible improvements to the process need to be explored. Automation of the system description step is one such improvement that should be considered. Based on real-world results, the tradeoffs described previously need to be either affirmed in their validity or reversed, if they prove to not be as beneficial as expected.

Tool support can be extended to accommodate more features related to the method itself (such as the option to use a grading- instead of a time-based system for change effort estimation) as well as quality-of-life features like a view dedicated to a specific goal such as system comparison.

Bibliography

[98] "IEEE Standard for Software Maintenance". In: *IEEE Std 1219-1998* (Oct. 1998), pp. 1–56. DOI: 10.1109/IEEESTD.1998.88278 (cit. on p. 13).

[ABP12] V. Andrikopoulos, S. Benbernou, M. P. Papazoglou. "On the evolution of services". In: *IEEE Transactions on Software Engineering* 38.3 (2012), pp. 609–628 (cit. on p. 24).

[Arn96] R. S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996 (cit. on p. 15).

[BG04] M. A. Babar, I. Gorton. "Comparison of scenario-based software architecture evaluation methods". In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE. 2004, pp. 600–607 (cit. on pp. 17, 27).

[BLBV04] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet. "Architecture-level modifiability analysis (ALMA)". In: *Journal of Systems and Software* 69.1-2 (2004), pp. 129–147 (cit. on pp. 17, 19, 23, 37).

[CHK+01] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, W.-G. Tan. "Types of software evolution and software maintenance". In: *Journal of software maintenance and evolution: Research and Practice* 13.1 (2001), pp. 3–30 (cit. on p. 13).

[CPDM16] B. Costa, P. F. Pires, F. C. Delicato, P. Merson. "Evaluating REST architectures—Approach, tooling and guidelines". In: *Journal of Systems and Software* 112 (2016), pp. 156–180 (cit. on pp. 28, 32).

[Gro+09] S. M. W. Group et al. "SOA Manifesto". In: *SOA Symposium, Rotterdam, The Netherlands*. 2009, p. 3 (cit. on pp. 11, 12).

[KABC96] R. Kazman, G. Abowd, L. Bass, P. Clements. "Scenario-based analysis of software architecture". In: *IEEE software* 13.6 (1996), pp. 47–55 (cit. on pp. 16, 29).

[KBAW94] R. Kazman, L. Bass, G. Abowd, M. Webb. "SAAM: A method for analyzing the properties of software architectures". In: *Proceedings of 16th International Conference on Software Engineering*. IEEE. 1994, pp. 81–90 (cit. on pp. 16, 29).

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere. "The architecture tradeoff analysis method". In: *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*. IEEE. 1998, pp. 68–78 (cit. on p. 16).

[LC02] W. Lloyd, S. Connie. "PASA: A Method for the Performance Assessment of Software Architectures". In: *Proceedings of the Third International Workshop on Software and Performance (WOSP'2002), July*. 2002, pp. 24–26 (cit. on p. 17).

[LS98] M. Lindvall, K. Sandahl. "How well do experienced software developers predict software change?" In: *Journal of Systems and Software* 43.1 (1998), pp. 19–27 (cit. on p. 12).

[MBZR03] T. Mens, J. Buckley, M. Zenger, A. Rashid. "Towards a taxonomy of software evolution". In: *Proceedings of the International Workshop on Unanticipated Software Evolution*. LAMP-CONF-2003-005. 2003 (cit. on pp. 14, 25).

[New15] S. Newman. *Building microservices: designing fine-grained systems*. Ö'Reilly Media, Inc.", 2015 (cit. on p. 13).

[OAS06] S. OASIS. *Reference Model TC,"Reference model for Service Oriented Architecture 1.0*. 2006 (cit. on p. 11).

[OW14] J.-P. Ostberg, S. Wagner. "On automatically collectable metrics for software maintainability evaluation". In: *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*. IEEE. 2014, pp. 32–37 (cit. on p. 7).

[Pap08a] M. Papazoglou. *Web services: principles and technology*. Pearson Education, 2008 (cit. on p. 12).

[Pap08b] M. P. Papazoglou. "The challenges of service evolution". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2008, pp. 1–15 (cit. on pp. 12, 14).

[PPR03] M. Polo, M. Piattini, F. Ruiz. "A methodology for software maintenance". In: *Advances in software maintenance management: technologies and solutions*. IGI Global, 2003, pp. 228–254 (cit. on p. 7).

[Raj97] V. Rajlich. "A model for change propagation based on graph rewriting". In: *Software Maintenance, 1997. Proceedings., International Conference on*. IEEE. 1997, pp. 84–91 (cit. on p. 15).

[Swa76] E. B. Swanson. "The dimensions of maintenance". In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press. 1976, pp. 492–497 (cit. on p. 13).

[Tre15] T. Treat. *"Designed to Fail" www.bravenewgeek.com/designed-to-fail/ . Retrieved 2018-11-14*. 2015 (cit. on p. 12).

[WGEZ16] N. Wilde, B. Gonen, E. El-Sheikh, A. Zimmermann. "Approaches to the evolution of SOA systems". In: *Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures*. Springer, 2016, pp. 5–21 (cit. on p. 12).

[WHHC14] S. Wang, W. A. Higashino, M. Hayes, M. A. Capretz. "Service evolution patterns". In: *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE. 2014, pp. 201–208 (cit. on p. 22).

[Woo12] E. Woods. "Industrial architectural assessment using TARA". In: *Journal of Systems and Software* 85.9 (2012), pp. 2034–2047 (cit. on pp. 8, 15, 20).

[WYZ10] Y. Wang, J. Yang, W. Zhao. "Change impact analysis for service based business processes". In: *Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 1–8 (cit. on pp. 12, 15, 21, 29).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature